AD-A047 595 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
A PROGRAM WRITER.(U)
NOV 77 W J LONG
N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-187 NL

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

12

MIT/LCS/TR-187

A PROGRAM WRITER

William J. Long

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-187
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): A Program Writer.
5. TYPE OF REPORT & PERIOD COVERED: Ph.D. Thesis, August 1977
6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-187

7. AUTHOR(s): William J. Long
8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
MIT/Laboratory for Computer Science
545 Technology Square
Cambridge, Mass., 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS:
Advanced Research Projects Agency
Department Of Defense
1400 Wilson Blvd, Arlington, Va. 22209

12. REPORT DATE: November 1977
13. NUMBER OF PAGES: 281

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
Office of Naval Research
Dept of the Navy, Information Systems Program
Arlington, Va. 22217

15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report):
Approved for public release; distribution unlimited

12 279p. 9 Doctoral thesis,

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
automatic programming
knowledge based systems
program design
automatic coding

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper is concerned with the problem of taking a high level specification for a program and designing an appropriate algorithm and data structure, utilizing knowledge about the domain and about programming. The basic approach is to use successive refinement organized and regulated by several models of different aspects of the programs. The system, called the programwriter, uses five models of the programs: as representing events in the application domain, as primitives passing arguments within a control structure, as creating and using data, as carrying on an I/O exchange with

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20.
the user, and as a construct in the target language. These models provide the appropriate views of the program to constrain and guide the refinement process. The global views provided by these models also make possible global transformations off the program structure when an opportunity for improvement is recognized.

MIT/LCS/TR-187

A Program Writer

William James Long

November 1977

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Number N00014-75-C-0661.

Massachusetts Institute of Technology

Laboratory for Computer Science
(formerly Project MAC)

Cambridge Massachusetts 02139

A Program Writer
by
William James Long

This report is a minor revision of a thesis submitted to the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology on August 31, 1977 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

This thesis is concerned with the problem of taking a high level specification for a program and designing an appropriate algorithm and data structure, utilizing knowledge about the domain and about programming. The basic approach in this thesis is to use *successive refinement* organized and regulated by several models of different aspects of the programs. The system, called the programwriter, uses five models of the programs: as representing events in the application domain, as primitives passing arguments within a control structure, as creating and using data, as carrying on an I/O exchange with the user, and as a construct in the target language. These models provide the appropriate views of the program to constrain and guide the refinement process. The global views provided by these models also make possible global transformations of the program structure when an opportunity for improvement is recognized.

Thesis supervisor: William A. Martin, Associate Professor of Electical Engineering and Computer Science

Acknowledgements

I would like to thank Bill Martin for his supervision, comments, criticisms and for being there with the important ideas when they were needed. I would also like to thank Gerry Sussman for many useful suggestions which contributed to the readability of the final document. Also, my thanks to Mike Hammer for reading and commenting on the thesis.

The other members of the Automatic Programming group have helped a great deal throughout the thesis work for which I am grateful: Lowell Hawkinson for the Owl implementation, Bill Swartout and Alex Sunguroff on the interpreter implementation, Bill Mark, Peter Szolovitz, Gretchen Brown, and Bob Baron for reading and comments on the various drafts.

For the support, encouragement, and help I needed to make it through, I thank especially my wife, Judy.

For making this experience meaningful and for leading the way, I thank the Lord.

Table of Contents

Acknowledgements . . . 3
Contents . . . 4
List of figures . . . 6
Chapter I Introduction . . . 7
0.1 The Problem of Automatic Programming . . . 7
0.2 My Approach . . . 9
0.3 The Example Problem . . . 10
0.4 The Need for Models . . . 13
Section 1 About Models . . . 15
1.1 What is a Model? . . . 16
1.2 What does a Model Do? . . . 18
1.3 How Many Models? . . . 21
1.4 What About Missing Models? . . . 24
1.5 What is the Relationship to Refinement? . . . 26
1.6 What are the Interactions? . . . 28
1.7 How are Models Implemented? . . . 30
1.8 What Happens in a Larger Domain? . . . 34
Section 2 An Example . . . 37
Section 3 Other Approaches . . . 44
3.1 Other Aspects of the Problem . . . 52
Section 4 Organization of the Thesis . . . 54
Chapter II How to Build a Programwriter . . . 55
Section 1 The Program Environment . . . 57
Section 2 Overview of the Programwriter . . . 59
2.1 Planning . . . 64
Section 3 Representation in Owl-1 . . . 66
3.1 The Capabilities of the Representation . . . 70
3.2 Relations . . . 74
3.3 Sets . . . 74
Section 4 The Specification Language . . . 76
Section 5 Time . . . 79
Section 6 Knowledge Base Structures . . . 81
6.1 Methods . . . 82
6.2 Schemas . . . 86
6.3 Intents . . . 89
6.4 Definitions . . . 91
Section 7 The Program Design Tree . . . 92
7.1 Goals and Ideas . . . 95
Section 8 The Service Modules . . . 96
8.1 Evaluation . . . 97
8.2 Whether . . . 102
Chapter III Models for the Programwriter . . . 105

Section 1 The Model of Design 107
1.1 Refinement 107
1.2 Control of the Design Process 109
1.3 Method Selection 111
1.4 Recognition and Modification 113
Section 2 The Domain Model 116
2.1 Interfaces with Other Models 125
Section 3 The Argument Passing and Control Model 126
3.1 Control in the Program 127
3.2 Iteration 131
3.3 Providing Arguments 132
3.4 Failure 136
3.5 Interfaces with Other Models 140
Section 4 The Data Model 142
4.1 Data Bases 151
4.2 Interfaces with Other Models 156
Section 5 The I/O Model 157
5.1 Interfaces with Other Models 165
Section 6 The Target Language Model 166
6.1 Implementing at the Lisp Level 167
6.2 Coding 171
6.3 Interfaces with Other Models 174
Chapter IV Scenario for a Savings Account Program 175
0.1 A Preview of the First Scenario 175
Section 1 The Basic Scenario 183
Section 2 Introducing Ideas for Efficiency 211
Section 3 A Variation on the Specification 219
Section 4 Efficiency Ideas for the Variation 225
Chapter V Selected Topics 233
Section 1 Answering Questions 233
1.1 Failure to Retrieve 235
1.2 Determining Change 237
1.3 The Size of a Set 238
1.4 Implications of an Answer 241
Section 2 Failure Revisited 242
Section 3 Ideas Revisited 245
3.1 Finding IDEAs 245
3.2 Implementing an IDEA 247
3.3 Comparing the Results 249
Section 4 Data Base Functions 250
Section 5 Philosophy of Programming 253
5.1 Goals Versus Recognition 254
5.2 Standardized Methods 256
Chapter VI Conclusions and Recommendations 259
Section 1 Mechanisms 259
Section 2 Models for Organizing Other Tasks 261
2.1 Models as an Aid in System Design 265
Section 3 Extensions 266
Section 4 A Last Look at Models 271
Bibliography 275

List of Figures

Figure 1. Model and program 19
Figure 2. Implementation of models 33
Figure 3. Refinement from datum to implementation 40
Figure 4. Program design tree, an example slice 42
Figure 5. Programwriter overview 60
Figure 6. Analyze and Plan 62
Figure 7. Analyze Algorithm 63
Figure 8. Time sequence of events 79
Figure 9. Form of program design tree 93
Figure 10. Detail of a program design tree 94
Figure 11. Evaluation by event inheritance 99
Figure 12. Model Interfaces 106
Figure 13. Refinement of an event into a datum 118
Figure 14. Control structure in the design tree 128
Figure 15. Searching for arguments 134
Figure 16. Failure algorithm 139
Figure 17. Datum sources and uses 144
Figure 18. Basic storage model 146
Figure 19. I/O view of a program 160
Figure 20. Initial nodes 176
Figure 21. Sketch of ACCEPT 178
Figure 22. Sketch of STATE 179
Figure 23. Model influence in program design tree 182

Chapter I
Introduction

0.1 The Problem of Automatic Programming

For a long time it has been realized that the major cost in using computers is in producing the software rather than the hardware. Furthermore, since each piece of software must be individually hand crafted by a programmer, the cost of software is rising while the cost of hardware is falling. A reasonable answer is to automate as much of the process of software production as possible[1]. This effort, called automatic programming, has a lengthy history and has resulted in many different tools to aid the programming process including compilers, higher level languages, and domain specific systems. While these tools have all aided programmers, programming is still with us, demanding much time and effort.

Programming is a multifaceted process extending from the acquisition of the problem, through the discovery of the relationships between the problem and the environment, the design of algorithms and structures, the production of code, the debugging of faulty solutions, the modification of existing code, to the gaining of expertise from completed efforts[2]. Programming also takes place in many different domains on different scales with different parts of the process taking on expanded or diminished importance. The problem of programming is so broad that any particular effort in automatic programming has little hope of achieving a general solution to the whole problem. Efforts thus far have achieved a certain measure of success by limiting

---

[1] Another answer is to standardize requirements for software, which is happening in ways such as the use of "turnkey" systems in business applications.

[2] For another view of the phases of the programming process see [Balzer 1972] pg. 17.

either the facet of the process or the domain of application and usually both. This is proper, but it is necessary to know what is general about the methods used and what is limiting. That is, what aspects of a method make it difficult to use in another programming situation? What additions are needed to handle a slightly different or more complicated or larger problem?

This thesis is concerned with *designing programs*, often called automatic program synthesis. That is, the facet of programming that takes a high level specification of the desired set of programs[3] and utilizing knowledge about the relationship between the programs and the domain and knowledge about programming, designs suitable algorithms and data structures, transforming the specifications into a set of programs that efficiently reflect the specification. Other projects have had a similar direction. The first systems claiming to do automatic program synthesis were based on automatic theorem provers, and operated by producing a constructive proof that the program's output specification, represented in predicate calculus, could be realized from its input specification. Other systems such as Hacker [Sussman 1973] and Mycroft [Goldstein 1974] have produced programs by debugging "almost right" programs until they work for the existing circumstances. Recently, the PSI project [Green 1976], utilizing a rule based system to drive a refinement process, synthesizes programs that handle various kinds of manipulation involving symbols. Many other projects could also be described as designing programs[4], making it important to distinguish the approach of this thesis and the characteristics of programming problems handled by this approach from those of other efforts.

---

[3] A set of programs is emphasized because of the need to understand and control the interactions between programs that share data -- an important facet of programming not included in most automatic programming problem domains.

[4] Section 1.3 will go into more detail comparing the various approaches to the automatic program synthesis problem.

0.2 My Approach

The basic approach in this thesis is to use successive refinement organized and regulated by several models of different aspects of the programs.

Starting with the problem specification, the *program synthesis system* develops the state of the program design in small refinement steps until the program is specified in the target language. A refinement step takes a requirement at the current level of abstract description and designs a less abstract way of satisfying the requirement. The particular requirement may be part of the algorithm structure, part of the data structure, or part of other model related structures. The result of the step may be more detail in an algorithm, a more specific statement of an operation, or an implementation of data closer to the target language.

Refinement is not a sufficiently powerful principle for organizing program synthesis. At any stage in the programming process there are many possible refinements that could be made. In effect, the search for refinements is a plausible move generator. The design process requires some way of controlling the selection of these refinements. Besides the problem of too many refinements, it may be necessary to make additions to the algorithm structure and data structure not provided by the refinements. For example, attachment of an initialization phase may simplify a program that might be executed later. Sometimes it is desirable to change an existing structure, because it is not always possible to anticipate the correct way to proceed at each step of the design.

The additional mechanism in this *programming by step-wise refinement* that provides the needed organization is the use of *models* to represent the various aspects of the programming problem from different points of view and provide the connections

necessary to handle these aspects in a natural way. A model is the representational structure for a view of the programs. For example, one might view the program as a set of operators passing arguments within a basic control structure. One alternative view of a program is as a set of I/O handlers conducting interactions with the user. Another alternative view is the set of programs as creating and using data. Different models provide different kinds of information and constraints for the design of the program. Their interactions guide the design process. The use of models to organize the refinement is the primary theme of the thesis.

0.3 The Example Problem

Before discussing the models and refinement in greater detail, it is necessary to specify the problem to serve as an example for these methods and introduce the system that will demonstrate the ideas. The system is called the programwriter[5]. It is written in Lisp utilizing a knowledge base written in a language called Owl-I[6]. The input to the programwriter is a specification in Owl-I for a set of programs. This specification will be turned into a structure in Owl-I that will record the advancing state of the design. The easiest way to describe the programming problem is to consider an example illustrating the problem. (Assume that the programwriter will handle the role of the programmer.)

A programmer is asked to write a set of programs to handle a very simple savings account system. The specifications for the problem include various facts: there

[5]A preliminary version of the programwriter exists which handles a simpler problem than is described here. It has been sufficient to test the mechanisms and understand the problems involved. There are no problems anticipated in implementing the version described here, other than time.

[6]Owl-I is a form of the Owl language which diverged from the main stream of Owl development. The essential features of the language and the knowledge base structures will be explained later.

will be deposits made to individual savings accounts; there will be withdrawals from them; and the user will occasionally want to know the balance of an account[7]. The specifications do not include information about figuring out the balance, what parts of a deposit or withdrawal must be included in the program, or the relationships between deposits, withdrawals, and the account. These the programmer can determine from more general knowledge about the domain and from the other information in the specification. The programmer has several tasks to handle. He must figure out what *balance* means in this context, how it might be computed, and what will be needed to compute it. He needs to determine what facts about the deposits and withdrawals will be needed for the programs and how to get them. He must select some way of interacting with the user to transfer the desired information. He must determine data structures and algorithms that will efficiently provide the programs with the capabilities required of them. To make these determinations involves a continual process including analysis of the situation, proposal of representations and algorithms, and refinement or modification of the proposed representations and algorithms. It may mean some trial and error. Eventually the programmer has to work out the representation of the programs in the primitives of the programming language. The ultimate result of the programmer's efforts will be programs written in a standard programming language (in this case Lisp).

The general programming problem would include variations on this example, such as additional programs to correct account information, restrictions on the execution order of the programs and restrictions on the possible inputs. It would also include sets of programs for other simple situations, such as a set of programs to sell tickets to theater performances and take information about new performances coming up. The general problem might be characterized as a fixed set of simple programs where each

[7]Variations of this example will be utilized throughout the thesis.

program either represents some corresponding event or answers a question simply computable from the events that have taken place. Several features distinguish these programming problems from programming synthesis problems tackled by other systems. (A few have some of these characteristics, none have all.)

1) The problem includes several programs which will be executed repeatedly over a loosely constrained time history, interacting with each other, and requiring the storage of information between program executions.

2) The programs must handle simple I/O interactions with the user, typing out questions and statements on the console in an effective way and reading the replies, detecting and handling any errors.

3) The specification is not complete, relying on the general knowledge of the programmer about the domain and programming to fill in the parts that are missing.

4) The programs produced should be efficient in utilizing the time and space resources.

5) The programer completely designs each program from the primitives of the system, not having existing programs available that might be modified to work.

6) The programmer has to deal with symbols, numbers and computations involving numbers at a simple level. Comparing basic alternative methods of computation is part of the problem, but doing complex symbolic mathematical manipulations is not.

7) The programmer does not have to interact with the user to get the problem specification, nor does he have to debug programs or learn from previous efforts. These are important parts of programming but they can be factored out of the process without diminishing the thrust of the thesis.

The problems are more general than those of other program synthesis systems in the sense that they require the programmer to handle a wider range of goals during the design. The "peripheral" issues of I/O, data interactions among programs, error handling, and so forth have not been factored out of the problem. Thus, these different aspects of the problem can interact with each other to introduce constraints on the solution and

require manipulations that utilize the capabilities provided by a multi-model view of the programming problem.

0.4 The Need for Models

The basic contention of this thesis is that the programming problem requires several different models of the programs, the program parts, and the relationships existing between the programs and their environment to express the relationships important to program design and make the problem manageable. That is, there are aspects of the problem that need different basic representations to capture the properties important for programming. The I/O interactions need to be expressed in terms of an I/O model of the program behavior; the requirements of the program parts for arguments need to be expressed in terms of a model of the argument and control relationships; the creation and usage of data needs to be represented in terms of a model of the data history; and so forth. This is an example of the situation described by Minsky[8]:

> "Sometimes, in 'problem solving' we use two or more descriptions in a more complex way to construct an analogy or to apply two radically different kinds of analysis to the same situation. For hard problems, one 'problem space' is usually not enough!
>
> "Suppose your car battery runs down. You believe that there is an electrical shortage and blame the generator.
>
> "The generator can be represented as a mechanical system: the rotor has a pulley wheel driven by a belt from the engine. Is the belt tight enough? Is it even there? The output, seen mechanically, is a cable to the battery or whatever. Is it intact? Are the bolts tight? Are the brushes pressing the commutator?
>
> "Seen electrically, the generator is described differently. The rotor is seen as a flux-linking coil, rather than a rotating

[8] In the paper on "frames" [Minsky 1974] pp. 53-55.

> device. The brushes and commutator are seen as electrical switches. The output is current along a pair of conductors leading from the brushes through control circuits to the battery.
>
> "We thus represent the situation in two quite different frame systems."

In writing a program there is a similar kind of situation. The programmer has to consider each problem that arises during the design process in terms of the model (or *frame or view -- I prefer* the term model) that is appropriate to the problem.

Consider what happens in the example programming problem when designing the program to handle a deposit. In terms of data a program or part of a program can create it, change it, or use it. This program will be the source for the deposits used by any of the other programs. That means it will handle the creation phase of the history of a piece of data representing a deposit, implying several requirements for the program. If there are uses of the deposits in other programs, the program will have to include a section to store the data representing the deposit in some form of long term storage. If the deposits are to be stored they will have to be in a form that can be handled by all of the programs using the deposits and will have to contain sufficient information to meet the requirements of the programs using them. Looking closer at the creation of the deposit datum, there are several phases: the information comes from the user; it is tested; and finally, it is accepted as a complete and proper deposit.

In terms of the flow of control and the passing of arguments there is another story taking place during the design of the program. At one level, the program has two parts: one (part A) that reads from the user, thus taking no arguments; and one (part B) that stores the deposit returned by the first part. Since part B takes as an argument the result part A, the order of the parts is constrained and part B must be after part A.

At a lower level, part B, which stores the deposit, is composed of subparts to access the data base, add the new deposit to the data base, and reset the data base to its updated form. The first of these takes no arguments. The second, however, takes two arguments: the data base retrieved by the first and also the deposit returned by part A. This discovery has refined the understanding of the way the result of part A is an argument to part B. As a result the order constraint has been relaxed on the subparts of part B, eliminating the requirement that the accessing of the data base take place after part A of the program.

These two models are concerned with the same parts of the program, but in different ways. Because of the different views, the concerns are different, the constraints are different, the terminology is different and the implications for the program are different. Both views of the program are important in determining the final structure and even other views must be considered if the finished programs are to satisfy their objectives. Each model is used to coordinate the design of some aspect of the program, to make sure it is implemented effectively and consistently.

Section 1 About Models

Granting that models exist, there are several questions about models that must be addressed. What constitutes a model? What does a model do for programming? How many models are there? What happens if models are missing? What is the relationship of refinement to models? How do models interact? How is a model implemented? What happens to the models when the domain becomes larger? These questions will be addressed at a general level in this section and will receive more specific attention in terms of the programwriter in later chapters.

1.1 What is a Model?

Models exist in more than just the domain of programming. In an arbitrary problem space a model is a view of the elements (or some significant subset) in the problem space organized around some common factor. The model relating the elements exists independent of the particular set of problems to be handled. It exists as a means of expressing relationships with respect to the common factor. The model has a set of rules about the relationships between the elements and a set of terminology to classify the elements. What kind of help the model can provide for a particular set of problems depends on the domain and the model.

Consider the following examples of models:

Problem: diagnosis of car troubles

Models: 1)The car is an electrical system. 2)The car is a mechanical system.

Problem: medical treatment of a person

Models: 1)The person is a circulatory system. 2)The person is a functioning being living in society. 3)The disease organism is a system affected by its environment.

Problem: designing programs

Models: 1)The program is a handler of data. 2)The program is a set of primitive operations passing arguments to each other.

By examining the examples, we can see several features of a model. Each model defines a closed space in the problem space. That is, the relationships of the model only rely on the aspects of parts that are in the model. That is why many of the models are called *systems*. This property means that a model defines a space in which problems can be solved. Secondly, there is interesting behavior within the model for the

problem domain, such that some useful chunks of the desired problems can be solved inside the model. This may mean for a particular problem that once the appropriate model is found, the solution process will be entirely in the domain of that model, or that several models will be able to make contributions leading to the solution. Finally, each model is part of a domain of expertise independent of the problem area. For example, viewing the car as an electrical system is part of the larger domain of electrical systems, about which much is known. The knowledge about the more general domain can be used in the specific model. Thus, the model is a way of focusing the knowledge from a domain of expertise on a problem area.

The programwriter has two kinds of models. To carry out the design process, there is a model of how program design takes place. This model is embedded in the basic control structure and drives the processing from specification to final program. To organize and regulate this design process, there are models of the programs. The program models provide views of the program as exemplifying particular kinds of system, such as a data handler, a representation of domain events, or an entity in the target language. In general, a model is made up of a set of associated interactions that can be understood in terms of some common factor. Thus, the parts of the program involved in handling data, for example, can be viewed as a substructure of the problem organized around the data requirements. Because of this substructure the factors required by each of these views can be used to keep the design process directed toward a program that will be reasonable according to each of these views. The model provides a framework of terminology and relationships with which to view the parts of concern to the model (which may include all of the parts of the program, but only a particular view of their use). The terminology identifies the important (relative to the model) parts of the program structure and the characteristics with which to differentiate between the

situations important to the model. The relationships provide constraints on the possible refinements, introduce requirements for missing parts of the programs, and help in the recognition of opportunities to transform the program structure. Consider the following simple model of programs as creating and using data:

> A datum is an entity created by a program and used by one or more programs. Its *source* is the step in a program where it is *created*, and its *uses* are any steps in programs that reference it. There are two constraints. First, it must be created *before* it can be used. Second, it must be stored when it is created if it is to be used in a later program execution.

This is a simple model, but it can have several different effects on the design of a program if it is used appropriately. If there is a use of a datum, there must be a source. If no source exists one must be created. Given a source, the execution of the source must occur before the execution of any use. If the source and the use are in the same program execution, the "before" requirement is a constraint on the order of the program parts. If the source and use occur in different program executions (either different programs or different executions of the same program), the execution of the source must occur before the first execution of a use or there is an error condition. If the source and use are in different program executions, the datum must be stored in some way. In such a case consideration must be given to the design of storage at some appropriate time during the design process. If the information will have to be stored, then the source will have to do the storing and the use will have to do the retrieving. And so forth...

1.2 What does a Model Do?

A model provides a kind of "global" view of the problem space (see figure 1). It provides relations and constraints that connect a subset of the elements of the

problem space around the common factor on which the model is based. The view is global (or at least more encompassing) because the elements are connected by a small number of associated relations into a system that can be understood in larger chunks (i.e., more globally) than could the raw elements with all of the different kinds of information they may have. Thus, a model gives to a decision point of a problem information that is global with respect to the decision point. The existence of different models means that a particular element can participate in more than one view of the problem situation and thus be subject to requirements from more than one model.

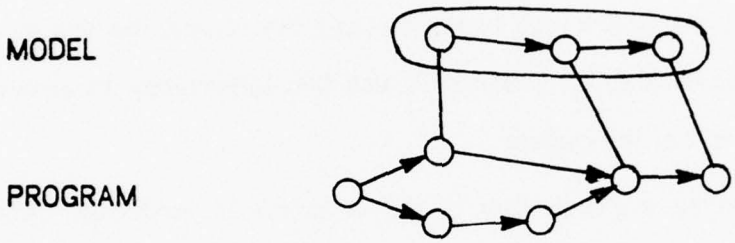

Figure 1. Model and program

For the programming situation the models of the program must provide the information to choose among the design possibilities. Using the information provided by the models guarantees that the program will be consistent with respect to the view of the model. Different models provide different global views, but each model provides a framework inside of which there are rules for consistency, reasonable tactics for design, and means for recognizing the important features of the situation. A model specifies a connection of the parts of the programs within this framework and provides an evaluation of the state of the programs in terms of the model. Use of a model may reveal parts of the program structure that are missing, places where errors can occur (and therefore must be where code is introduced to anticipate them), and provide

characterizations of the programming situation needed for other parts of the programming process. A model also provides a terminology in which the procedures and facts for doing programming may be written. The procedures and facts need this terminology to specify the program parts to which they can be applied and the characteristics which they use. To refer to the program parts in terms of the model, the system must be able to handle such references. For example in the model above, the three terms *source*, *use*, and *before* identify parts and relationships that link the program parts to the model's facts. The procedure for assuring that the constraints between these parts are handled must be written using those terms. To make use of such a procedure, it is necessary to locate the place in the code that represents the *source*, the *use* or answer questions about the *before* relation. The models use this terminology to communicate their information to the rest of the system.

Besides providing a global view of the programs to guide the refinement process of the programwriter, the models also are an indication of the capabilities of the system. That is, if the system is capable of handling the requirements of certain specific models, a description of the models can be used as a description of the capabilities of the system. This is true not only of the programwriter, but it is true of any system that uses models for organization. This is important in large systems where it is difficult to convey to a user what the system can and can not do. The user is capable of thinking in terms of models and can infer from the model what kinds of situations the system should be able to handle and even what a particular construct will mean to the system. For example, if the programwriter is described as handling I/O with the user to ask for information or state results, the user concludes that the system can formulate reasonable questions, print them out on the console in some understandable format, read the reply, and use it as the answer to the question. The user would be disappointed if the system was not able to handle these things.

1.3 How Many Models?

There does not exist a fixed list of models that will provide suitable frameworks for all of the circumstances that might be encountered in an arbitrary programming situation. Some models are needed all of the time, while others are only useful in specialized circumstances. This is true of all complicated problem solving areas. Again, consider the automobile example: In a race car, one of the considerations is the stability of the car. To properly understand what might be happening, the car must be viewed as an aerodynamic object. The shape of the car creates a certain amount of lift, releasing the rear wheels from the ground unless it is properly compensated for by "spoilers". This view of the car as an aerodynamic object is a legitimate model. It is just limited in its usefulness. The normal mechanic, working on passenger cars would not have to understand this model to be effective in his work. This is a case where the specific instance of the general model of aerodynamic objects has enough to say about the problem domain to be useful. Similarly, in the programming situation some models are more useful than others or more useful at certain stages of the programming process. In a real-time programming situation, viewing the program as a *user of time* might become an important model involving the circumstances under which it would use more or less time and typical time usage patterns, but in the normal programming situation a simpler view of execution time as a parameter to be optimized is sufficient.

Different models also have different domains. That is, they take into account different parts of the programming problem. In the automobile example the mechanical model is useful while working with any of the parts of the car, but the electrical model is only useful while working with the parts included in the electrical system. In the programming situation, the control and argument passing model is concerned with all of

the parts of a program and how control and results can be passed between the individual operations. On the other hand the data history model, while having a more global outlook on the programming situation, is only concerned with the parts of the programs that create, use, or change data. Furthermore, the I/O model is only concerned with those parts of a program that do reading or printing.

For programming in the domain of the thesis a small number of models have been chosen which adequately cover the important considerations of the programming domain and demonstrate the implications of organizing the programwriter around models. These models represent several different aspects of programming such as the relationship between the programs and the things they represent, programming as problem solving, and the details of programming language considerations. The models will be incorporated into the system to control the refinement and guide the design. They will be developed to varying degrees of complexity depending on the needs of the programming domain. The models are as follows:

1) A model of the programs as abstractions of the real world events they represent. The running of the programs represent events in the real world, such as depositing money into a savings account at a bank. Thus, a model of the programs as abstractions of these events is needed to maintain plausibility. That is, the model can provide the defaults and restrictions that insure the programs are valid abstractions of the events they correspond to. This model is also the source for any definitions of the terms in the specification that might be needed to design the program.

2) A model of a program as a set of primitives connected through argument passing and the basic control constructs such as conditionals and iteration. This model emphasizes the minimal order constraints needed to get the program to do what it is supposed to do. It is not concerned with the ultimate linear ordering of the target language primitives. It also takes the view of the program as a problem to be solved by properly connecting the primitives.

3) A model of the programs as creating and using data. The creation and use of data is very important in a program system

such as this, and represents a model that in many ways controls the design of the programs. This model takes a more global view of the set of programs, considering possible execution histories. It is responsible for making sure the programs are prepared to handle the data in whatever states are possible within the constraints on the usage of the programs.

4) A model of the I/O interactions between a program and the user of the program. This model maintains the view of what will appear on the user's console, that is, how the program will react for the user. It is responsible for the form and content of the output to the user and for securing and validating the input from the user.

5) A model of a program as a structure in the Lisp program formalism. This is the model that handles local variables, the format of special Lisp constructs, and the general format of the program. It is responsible for the final determination of the order of the primitives.

These models are all useful in most kinds of programming situations. Different programming situations would require these and perhaps more models (or at least more extensive versions of these models). At first it would appear that the I/O model would only be needed for programs that do I/O with the user, but in reality that model deals with the exchange of information between the programs being designed and their environment. Any set of programs having more than a trivial amount of interaction with its environment would require a similar kind of model in the designing. If fact, most programs have other kinds of I/O to handle and would require a more extensive I/O model. For some areas of the thesis domain the models will be more developed than for other areas. The characteristics of the domain require that the handling of failure and the format of simple in core data bases be well developed. These could be separated out as their own models, but there is no need to do this as they are really parts of other models, in the same way as the model for the electrical behavior of the generator is part of the electrical model of the car.

1.4 What About Missing Models?

Because some views of the programs are not important or not difficult to handle in the programming domain, it is possible to get by without models for these views. This brings up the question of what the alternative to a model might be. A model provides an organization for the ways in which the state of a design may be advanced. Without the model, the system must still have some way of controlling the process. How this happens depends somewhat on the circumstances. Basically the model must be limited sufficiently to fit within the view of another model or set of models, possibly augmenting the existing model to take into account important facets of the missing model. For example, a limited view of program resource usage is a weighted sum of the average time and space it uses. For most purposes this view is adequate. This view is sufficiently restricted that it will fit within the data model and the argument and control model. The data model will provide the information necessary to determine the amount of space required, and the argument model will provide the amount of time required. The result of eliminating something like the I/O model would be a set of parameterized routines (macros) that can be handled within the framework of the argument and control model. The macros can be inserted when needed and vary depending on factors that can be checked in the existing environment. For example in the programwriter, the I/O could be handled by a set of parameterized routines. For the routines to provide reasonable I/O they would have to be designed to produce unambiguous statements no matter how they were used and the results would have to be indivisible, preventing other parts of the system from reordering the print and read steps. Even at that, undesirable situations could arise unless some additional functions of an I/O model were handled somewhere in the system. The order of the I/O

statements may be inconvenient for the user, or may include redundant information. Without the model the I/O image the programs give the user is the result of the canned routines, which were not designed with the particular program in mind, and the results of the decisions made during the design of the program for reasons separate from the effect those decisions would have on the I/O.

The key difference between the use of models and the lack of models is a kind of "global view" provided by the models. The model, because it is concerned with a limited number of properties of the program and because those properties and their effects on one another are understood outside of the context of the particular program and this understanding is incorporated into the model, can provide a global view of the thread of those properties through the programs. Thus, the models provide part of the answer to Sussman's complaint about his own system, that it lacked an "overview" of the programs[9].

Another way to look at models is that they know the kinds of information they will need to make decisions, the kinds of decisions they need to make, and how to represent the results of those decisions. The alternative is a parameterized system which is limited to those alternatives that can be represented as parameters and to decisions based on information that can be gathered within a less specific framework. Statements like "If any of the programs will need the datum in a later execution, include code at the source to store the datum" is outside the realm of any obvious parameterization.

---

[9][Sussman 1973], p. 184.

1.5 What is the Relationship to Refinement?

The use of models to organize the programming design represents the application of *strategies* to the refinement process. That is, they provide a methodology for choosing among the alternative refinements. As Davis states[10]:

> Faced with a PKS set [set of applicable chunks of knowledge] of non-trivial size, some decision must be made about which KS [chunk] should be the next to be invoked. This is exactly the task which strategies are intended to confront. It is our contention that this decision point is an important site for the embedding of knowledge, because system performance will be strongly influenced by the intelligence with which the decision is made. We claim also that it is a decision which in many systems is made on the basis of knowledge that is neither explicit, nor well organized, nor considered in appropriate generality.

The situation in programming design by refinement is a little different from the situation Davis is confronted with. He has many possible chunks of knowledge at any point -- here there are a few, but he can tell sooner if the correct one was chosen. Therefore in terms of cost for deciding what piece of knowledge to use next, the situations are comparable. For the programwriter the models provide these strategies that Davis is recommending. They suggest appropriate refinements based on the needs of the model and generally work to coordinate the refinement process.

There is however a key difference between strategies as Davis sees them and the use of models in the programwriter. Strategies are an add-on -- available in case a selection will be time consuming. They take a given node with a set of applicable rules already generated and provide mechanisms to help choose among them. The basic view of the strategy seems to be local to the node. That is, it may exist because of some higher node or some more general information, but its action is local to the node in

---

[10] In his thesis on meta level knowledge for knowledge-based systems [Davis 1976] pg. 201

question rather than coordinating several nodes. The kinds of determinations a strategy is to make involve ordering information for the applicable rules. The models on the other hand are organized around views of programs which can stand by themselves. Their views of the program provide not only strategies to choose among the possible refinements of a node, but also a coordination among the refinements of different nodes. The initiative for the actions is with the models. The models not only choose among existing possibilities but also create possibilities that would not otherwise have been considered. The most important fact about models is that the strategies they provide are organized on a global basis. That is, on the basis of a global view appropriate to the considerations of the model, and therefore appropriate to the potential actions of the node.

Davis also proposes a scheme for using the strategies to suggest other possible alternatives, but there is a difference between Davis' scheme and the way that models produce other alternatives. On top of the strategies are meta-strategies (strategies about using strategies). One of these might be to suggest that if more than one strategy wants to try a rule that was not among those that fit, the rule should be tried anyway. With models the other alternatives are proposed as a result of needs discovered by the models viewing the programs from a more global perspective, that would not have been discovered as a need local to the node. Because the models are able to take care of global needs this way, the refinements used by the programwriter can afford to be more specifically addressed to the nodes they are refining, without worry that more global considerations would indicate some refinement that would be left out.

The models provide refinement with an organization based on satisfying the

requirements of the different views of the program. These views have organized fields of knowledge associated with them, independent of particular programs. An expert programmer knows a great deal about the handling of data or the use of a particular programming language. This knowledge comes complete with its own terminology and can be discussed in its own right. When the expert applies his knowledge to a program design problem, he does so through viewing the program in terms that let this separate knowledge aid in the process. The programwriter, since it is organized around these views, has an appropriate structure to use the same kind of expert knowledge. That knowledge is used to restrict the refinements, suggest refinements that would not have been considered, coordinate refinements in different parts of a program and in different programs, and provide the characteristics needed to identify the situation.

1.6 What are the Interactions?

In a system with a number of models, there will be interactions between the models. Depending on the system, these may be weak interactions or strong interactions. If the important behavior in the problem space occurs within models, the primary interaction is just in finding the correct model in which to solve a specific problem. If the important behavior crosses model boundaries, the system must be prepared to pass information between models. These interfaces are the primary interactions between the models in such a system, but the models may interact in other ways. Each model makes assumptions about the environment in which it is operating and it is up to the system to keep those assumptions valid in order to prevent undesirable interactions.

The purpose of the models used in the programwriter is to guide the

refinement of the state of the design. The refinement process for any program crosses the boundaries of all the models in the system, making the interfaces between the models important. All of the models are involved in the effort, but not all at the same time. Each model has to be able to make its contribution when it has something to contribute and the result must be understood by any other models dealing with the same structures. Thus, there is a need to control the interactions between the models. For example, it may be discovered from the data model that code needs to be included to store some datum. Once that decision is made, the code must fit within the control structure under the scrutiny of the argument model. After the basic control structure has been determined, the target language model will have to fit the parts into the available Lisp primitives. Thus, it is necessary for the target language model to have some say over the selection of control structures.

The positive ways which the models interact are the use of terminology recognized by each of the concerned models, by models producing goals for other models to handle, and by providing routines to answer questions about the domain of the model of interest to other models. The models can benefit from common terminology because their results are kept in a form accessible to any other model that might need it. In a practical system, there will be situations that can be expressed in the common terminology that can not be handled by one or more of the models using that terminology. The potential for undesirable interactions of this kind must be handled in the design of the parts of the system. Either the models can be limited so they only produce constructs that are understandable by the other models or there must be some protocol for resolving differences. The interactions of models can also cause certain kinds of timing problems. For a model to be used, the state of knowledge about the programming situation must be consistent and complete at the level needed by the model. An

example is the use of the data model. The data model needs to know all of the uses of a datum to make appropriate decisions about it, however refinement may not have proceeded to a point where all of the uses are known. In such a case, the examination by the data model will have to be postponed until the usage knowledge is complete.

1.7 How are Models Implemented?

There are three choices for ways of implementing models: directly as a set of facts that are then used by an interpreter of some type to provide guidance in the problem solving, indirectly as a set of modules that provide the information from the model for the particular kind of problem solving situation, and indirectly as an organization of the system. The Metadescription System of Srinivasan[11] is a system that implements models directly (actually the system uses a single model, which would correspond to the domain model). The *structural knowledge* provides the terminology of the model. The *sense knowledge* provides the constraints. And, the *transformational knowledge* provides guidance for handling questions relating to the model. To use a model in this form, the system has three kinds of interpreters: a checker-instantiator to maintain the basic consistency of the model, a theorem prover to answer questions about the model, and a designer to use the model for actively generating requirements for the problem solving effort.

The difficulties with such a system stem from the low level nature of the information. The interpreters must be general in order to handle whatever model the system is given. This means the system itself can only indirectly take advantage of the

[11] The system is described in [Srinivasan 1973-1] and its possible use as the basis for an automatic programming system is described in [Srinivasan 1973-2].

organizational opportunities provided by a particular model. Nor can the system take advantage of the interactions known about a particular model in a particular problem space. One of the advantages of using models in the first place is all of the problem solving information that the model and its generalizations bring to a particular problem space. Such an implementation of models makes it difficult to include such information in the system, because it does not fit neatly into the three kinds of allowed information. Much of the important information says how the interpreters should operate, information that is easier to represent procedurally than declaratively. Consider a part of the argument model used by the programwriter: A piece of *sense knowledge* in the model states that every node requiring an argument must be connected to a node providing the argument. In the problem space of designing programs, this requirement can be turned into a procedure that checks for argument requirements, searches back through the program for a node that will satisfy the requirement, and either links the nodes if one is found or sets up a requirement to produce a new node that will satisfy the requirement. The corresponding actions in a system like Srinivasan's take considerably more work. The need for an argument is found by checking all of the sense knowledge and discovering that one requirement is not satisfied. Once the requirement is discovered, a solution is found by searching the transformational rules for rules that fit the difference between the desired state and the existing state. Unless there are additional transformational rules specific to the problem space about using the model rules, there is no guarantee that the correct rule will be found first.

The second and third ways of implementing a model offer efficiency and the ability to use the higher level problem solving help at the expense of the ability to change models and change problem spaces. Both of these ways may be considered

procedural embedding of knowledge[12]. Using the model information as a set of modules eliminates the searches that would be required in a more general system. To do so requires that the interactions that will occur between the sense information and the transformational information have been abstracted out of the model as needed in the problem space and the appropriate ways of handling these interactions have been turned into procedures. Thus, it is the actions that will result from the appropriate use of the model that are implemented rather than the model itself. For the programwriter, this is particularly important. The basic refinement model of program design does not consider the possible interactions between different parts of a program, but the program models determine these interactions and what additional kinds of information are required to control them. Thus, the models are reflected in the constructs used to represent the state of the design and pass information, goals, and requirements between the parts of the system.

Conceptualizing the problem space in terms of the models also helps to determine an overall structure for the system. The interactions of the models form information paths that are natural boundaries for the breakdown of the system. The structured programming approach is to turn the actions of the system into modules and refine them. However, in the case of refinement the models imply an additional level of control besides that of the refinement mechanism. By including the services of the models in the basic organization of the system, this additional level of control is made explicit, and the actions of the models more efficient. In a problem space that is well understood, the model can be transformed into an algorithm for solving the problem. This is the case for the problem of parsing many kinds of grammars[13]. That is, if the problem

---

[12] As described by Hewitt in [Hewitt 1972]

[13] See for example [Hammer 1974] where it is shown how to produce a system for parsing any LL(k) and LC(k) grammar.

were to parse a string in a given grammar, a system could be built by transforming the grammar (the model) that would solve the problem. Unfortunately, problem areas that are so well understood are few and far between. In most cases, the best we can hope for is to use the models and the services they will perform in the problem space as an aid in dividing the system into modules.

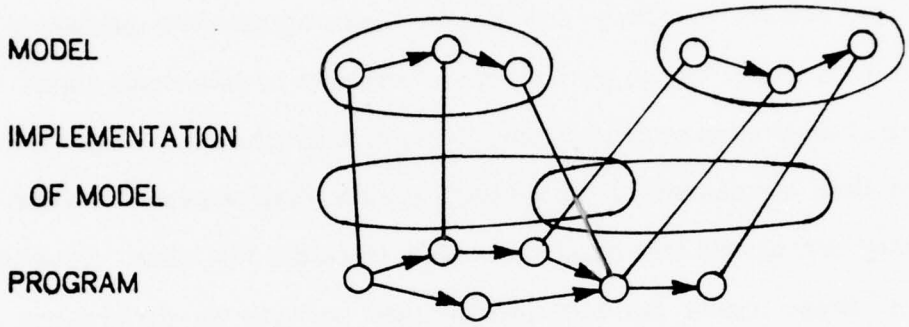

Figure 2. Implementation of models

The programwriter implements the models procedurally. The basic organization of the programwriter reflects the reliance on models in the modules that carry out the design functions and in the data structures that record the state of the design. (These will be discussed in detail in the next two chapters.) As indicated in the figure, the implementation is between the explicit models and the problem solving representation. The modules are tailored to the problem space to efficiently carry out the services that will be required in programming. On the other hand, the modules remain as separate entities to retain the flexibility of using them whenever the rest of the system needs the service. This arrangement represents a compromise which seems to be appropriate for the programming problem. For other kinds of problems where models can be used in conceptualizing the problem, the appropriate implementation of the models may be different.

Because the models are used to organize the refinement process, the modules carrying out the services of the models have been selected to fit inside the refinement framework. The possible uses of a model such as the data model occur in all facets of the refinement process, making it necessary to provide modules consistent with the system structure for refinement and with the requirements of the other models for services. The data model has a part in recognizing the kind of data situation, designing data structures, deciding on coding requirements, answering questions pertaining to the data, and finding parts of the program structure refered to in data model terms. These functions are divided up in accordance with the system structure into modules that can be called as they are needed. Thus, although together they represent a view of the programs, they are spread throughout the system to handle that view. Other models, such as the domain model can best provide their services as declarations in the knowledge base to be accessed whenever the information is needed. And, the target language model provides part of its services as the final refinements that convert the most concrete level of operations into code. There are also a few cases where services must cross model boundaries. An example (which will be discussed in chapter five) is counting the instances of data since some event. The data model provides the appropriate view to determine what might have happened between the desired program executions, while the argument and control model provides the appropriate view to determine what might have happened in a program before or after the desired event.

1.8 What Happens in a Larger Domain?

One issue that must be faced by any large system is what happens when more information is added to it. That is, can the system be expanded to handle larger

problems, or problems in different areas? And, what happens to the system when it does expand? Does the expansion only affect local areas in the system, or does it require global changes or a slowdown in speed? The claim is made that modularity simplifies the modification or extension of a large system. However, the way in which the system is divided into modules makes a fundamental determination of the ways in which the system can be flexible and therefore the ways in which the system can be extended. Changes within a module are easy (or at least proportional to the size of the module). Changes in the interfaces between two modules are harder, involving both of the modules. Changes in the overall relationships between modules are nearly impossible.

The programwriter too is divided into modules, only the boundaries along which it is divided are different from other systems. Other systems have divided the programming process and in turn their program synthesis systems into phases[14]. A proposal for four phases into which to divide the programming process is contained in [Balzer 1972]. These phases are problem acquisition, process transformation, model verification, and automatic coding. The division of the programwriter is not based on such phases, but on the models. These are more natural lines along which to make divisions. Internally they have a consistent view of the program. Externally, they can communicate with the other models through shared terminology. The influence of the models also accounts for the appearance of phases in the programming process. The early stages of the design involve the domain model to a large extent and its function in filling in the definitional details not explicit in the specification. This might be called an acquisition phase. (The programwriter has a very simple view of acquisition, but it could be extended, making this correspondence clearer.) After the specifications have been

[14] See for example the organization of the PSI system [Green 1976], which will be discussed further in the next section.

filled out, the refinement process, the argument model and the data model detail the basic algorithms and fill in missing operations. These are functions that could be called process transformation and model verification. Near the end, the target language model gains influence, doing the "automatic coding". The difference between the organizations is that the programwriter is divided according to the basic models of the program and programming, while a system divided according to Balzer would be divided into phases where particular models would *normally* have the greatest influence. Such a division rules out many of the interactions that models could provide. It also means that some of the functions of models will have to exist in more than one phase. If those particular functions are the ones that need to be extended, the division works against the orderly expansion of the system.

The orderly expansion of the programwriter is predicated on basic influence of the models. Inside a model there is a consistent view of the program from which the modules of the model draw their functions. Once the basic view taken by a model has reached a useful level of complexity (maybe twice that of the version described here?), the modules can be viewed as separate entities providing specific functions for the programming process. The interfaces between the models and hence the modules are provided by common terminology. This too is tied to the stability of the program views of the models. That is, the terminology is about relationships that exist between aspects of the program design and these have meaning as parts of the views of programs held by the models. One model can ask for services from another model using this terminology. The only knowledge the models need about each other is what constitutes legal syntax of the terminology. As the models become more developed, the syntax of the constructs they can understand becomes more complete and the amount that one model needs to know about another decreases. Thus, once the models have reached a

level of completeness, the interface terminology will also level off. This leaves the modules to grow in abilities and the refinements to grow in range. This is an optimistic forecast, but the fact that the models themselves constitute bodies of knowledge which can be separated from the programming process, lends credence to the view that this is the natural way to divide up the process. It also gives reason to believe that changes in views about one such model will not influence another model.

Section 2 An Example

To show how the programwriter goes about designing a program, this section will examine a small part of the process for the earlier savings account example. The problem includes three different programs, one of which accepts information about a deposit from the user. This program will be called when the user deposits some money to his account and will be responsible for providing that information to the rest of the system.

The program is initially represented as a node in the Owl-1 knowledge base. Call that node ACCEPT-DEPOSIT[15]. Exactly how that node fits in with the other parts of the knowledge base will be described later in the thesis. For now the important facts are that it is connected to information about accepting things from the user, about deposits, and about the other programs in the set.

The programwriter first examines this node by matching it against a structure called an INTENT which brings out some of the important implications and needs of the node. These are used to simplify the recognition process for other modules and to set

[15]The notation for this section is simplified to make it readable without introducing too many details. The more detailed version will be presented later.

up goals for the design. The INTENT of the ACCEPT-DEPOSIT node produces the following pieces of information:

ACCEPT-DEPOSIT
1)SOURCE for DEPOSIT
2)INPUT of DEPOSIT
3)must CHARACTERIZE DEPOSIT

That is, the program refined from this node will contain the point at which the DEPOSIT is created. (That point is called the SOURCE.) The program will receive the DEPOSITs as input from the user. And, the DEPOSITs will have to be handled by the CHARACTERIZE module before any real decisions about the structure of the program can be made.

To produce a program from this node, it will have to be refined. An algorithm will have to be developed that will take the input from the user, asking him for the parts that are needed. It will have to do something with the information from the user to make it available to the other programs that will need it. The DEPOSIT itself will have to be refined and turned into some more concrete structure that can exist in the target language.

While analyzing the rest of the program nodes, it is discovered that the calculation of the balance will make use of the DEPOSITs provided by this program. The calculation comes from the basic definition of balance provided by the domain model. It states that the balance is the sum of the deposits minus the sum of the withdrawals. This definition can be directly turned into an algorithm by performing the computation when the balance is needed. Without questioning the effectiveness of this way of calculating the balance, it implies that all of the DEPOSITs that have happened for a particular account are needed to calculate the balance of that account.

The programwriter will be ready to do the characterization of DEPOSITs

required by the INTENT once all of the initial program nodes have been analyzed. By then all of the nodes that use or affect the DEPOSITs will have been marked by the INTENTs. Characterization is handled by a planning module called CHARACTERIZE that is part of the embodiment of the data model. It is the process that determines the features of the data which will be needed to select the program parts and data base parts to handle the data. In this case the DEPOSITs can only come from the ACCEPT-DEPOSIT program and can only be used in the balance computation. This is a fairly simple situation, but it does require that the DEPOSITs be stored in some kind of data base for the later balance computations. The characterization module records this requirement by putting a note on the program node to ASSERT the new DEPOSIT at the appropriate time. (Notice that there must be cooperation among the various parts of the system, because that appropriate time is not yet known.)

Besides the information from the analysis, the characterization process uses information provided by the domain model. The nature of DEPOSITs is specified by a static definition, which is part of the domain model. The definition is a specification of all of the properties of a deposit that might be needed in a datum representing a deposit. That is, it is an abstraction of *deposit* as a possible datum in a program. From this generalized version, the parts appropriate to this programming situation can be selected to refine the DEPOSIT representation. The definition provides several pieces of information. A DEPOSIT is identified by the account that received it and by the time it took place. That is, two DEPOSITs are different if they occurred at different times or were for different accounts. Besides this identifying information, DEPOSITs can also have an amount associated with them and the name of the branch of the bank in which they took place. The characterization process is responsible for deciding how much of this information will be necessary for the representation of DEPOSITs in the programs. In this case only two parts of the definition will be needed.

DEPOSIT (as needed for programs)
ACCOUNT-NUMBER (identifier)
AMOUNT-MONEY (content)

The ACCOUNT-NUMBER is needed to identify the DEPOSIT sufficiently to use it in the correct balance computation. The AMOUNT-MONEY is needed to be able to do the computation. This determination of a semantic structure for the datum, starting with the form supplied by the domain model and tailored by the data model provides the necessary components for the program. This is the beginning of the design of a data structure. There will still be the selection of a target language representation, a data base to hold the actual items, and design of code to create and access the items. That is, the data will have to go through the stages of refinement as outlined in figure 3. The rest of these refinements will take place later in the design process when more is known about the programs.

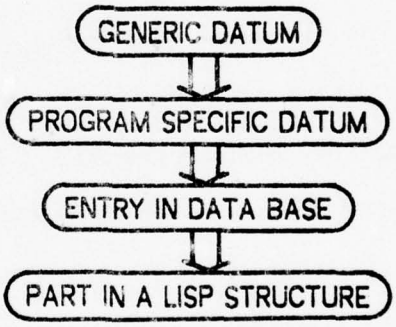

Figure 3. Refinement from datum to implementation

Once the characterization is complete, there is enough information known about DEPOSITs to choose a method for the ACCEPT-DEPOSIT node (that is, a refinement of the node in the usual sense). The method is chosen by matching the information known about the node against the requirements of the available methods. A method is a

statement of how to carry out the operation desired by the node. It consists of some small number of steps, each of which is slightly more concrete than the original node. The steps are connected by the basic control primitives of the argument and control model, specifying any constraints on the order of the steps. Because the method is a static structure, it must be instantiated for the node. The method chosen, after being instantiated, provides the following two steps:

```
METHOD for ACCEPT-DEPOSIT
    ASK for DEPOSIT
    THEN the (RESULT of ASK) BECOMEs a DEPOSIT
```

This method, besides specifying that it is necessary to ask the user for the deposit, contributes the following facts: the result of the asking step will be the DEPOSIT; it will only be a DEPOSIT after the asking step is complete; and after the asking step has successfully completed the DEPOSIT is complete and acceptable. The second step of the method is a point of interaction between different models. From the point of view of the argument model, it says "now I have a complete deposit ready for whatever should be done with it". For the data model, this point in the program is the SOURCE for DEPOSITs. This is the point at which any code to be "at the SOURCE of DEPOSITs" should be inserted, including the step specified by the characterization to ASSERT the DEPOSIT.

The first step of the method, to ASK for the DEPOSIT, is an entrance for the I/O model. To ASK for a DEPOSIT means interacting with the user to get all of the information needed by the program to make a complete DEPOSIT. The I/O model, in cooperation with the argument model will be used to design code that will neatly output appropriate questions to cause the user to provide the account number and amount of the deposit. The questions and responses will obey the conventions for format and interaction, such as putting a carriage return between interactions, putting a space between words, and not asking unnecessary questions.

At this point the program has two steps:

```
ACCEPT-DEPOSIT
      ASK for DEPOSIT
      ASSERT result of ASK as DEPOSIT
```

The next action to be taken by the programwriter is to the examine these two nodes by using the appropriate INTENT information. That will in turn set up more requirements for work -- for the I/O model to design the ASK code, for the data model to design a data base in which the result can be ASSERTed, and so forth.

The programwriter will continue in this fashion until it has refined the nodes to the point where they are directly implementable in the target language. That is, all of the operations that the program must perform are known, and they are known to the level of detail of the language in which the program will be written. Thus, the ASK code will include the necessary Lisp to print requests to the user for the desired information, read the replies, check to see that the replies are reasonable, and pass the results on to the ASSERT part of the program.

In this situation the refinement of the ASSERT part will result in the code necessary to put the information into a data base. Figure 4 gives one cut through the refinement of the ASSERT part (i.e., one path through the tree).

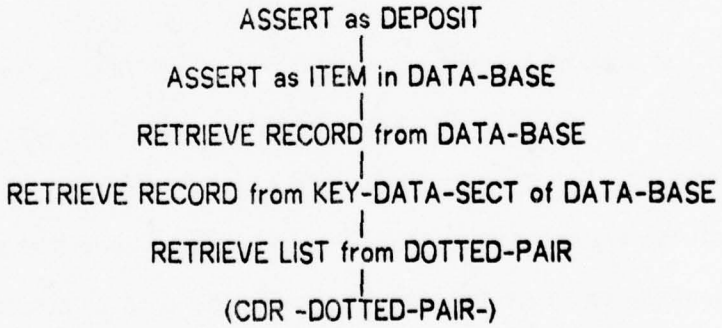

Figure 4. Program design tree, an example slice

The DEPOSIT is implemented as an ITEM in a data base, making the ASSERT become an operation on the data base. DATA-BASE has internal structure besides the ITEMs. The ITEMs are divided into groups based on the ACCOUNT the DEPOSIT is for, with all of the DEPOSITs to the same ACCOUNT in one RECORD. Each RECORD is in a KEY-DATA-SECT, consisting of the ACCOUNT and the RECORD, and so forth. The first step of ASSERTing an ITEM into such a data base is to RETRIEVE the RECORD it will become a part of. The last step of that operation, after the KEY-DATA-SECT that holds the RECORD has been retrieved is to RETRIEVE the RECORD from the KEY-DATA-SECT. All of these parts of the data base must be implemented as Lisp structures before code can be produced. In this case, the KEY-DATA-SECT is implemented as a DOTTED-PAIR of the ACCOUNT (as a KEY) and the RECORD. The RECORD is implemented as a LIST containing all of the ITEMs. Thus, the refinement of the operation is to RETRIEVE the LIST that represents the RECORD from the DOTTED-PAIR that represents the KEY-DATA-SECT. For this operation there is code available that takes the "CDR" of the DOTTED-PAIR.

Even after all of the operations are determined at the target language level, there is still some work to be done to decide on any unconstrained orderings and to satisfy any needs of the target language model in producing a proper program. These must be handled as a final step before a program can be produced.

This section demonstrated some of the processing that takes place in designing a program. For this processing to take place there is a great deal of structure that must exist in the programwriter. Explaining what these pieces are and how they fit together will be the subject of the next chapter.

Section 3 Other Approaches

The idea of automatic program synthesis is not new. There have been many projects that have attacked the problem directly in some form and many more that are related to the problem. There have been four basic approaches: the theorem proving approach, the debugging or evolutionary programming approach, the programming from examples approach, and the refinement approach[16].

The first of these, which has received the most attention and the earliest attention, is the theorem proving approach.[17] This approach is not based on any theory about the way people write programs, but rather an observation about proving theorems. The idea is that a program is a transformation of the input into the output. Therefore, a constructive proof that the transformation is possible produces as a byproduct a program for accomplishing it. The necessary prerequisites for producing a program by this approach include: a formal description of the input and output of the program, a formal description of the programming environment, and a theorem prover powerful enough to prove the output statement from the input statement. In actual practice the theorem prover may need additional help from the user to employ variables, establish loop invariants, and guide the goal structure.

This approach is appealing because the resulting program has an implied guarantee of correctness. The proof that produced the program is also a proof that the program does indeed do the desired transformation. Still there are a number of

---

[16]For a general overview of the different approaches to automatic programming see [Biermann 1976] and [Goldberg 1976]. [Heidorn 1976] also contains an overview, but emphasizes natural language specification.

[17][Manna and Waldinger 1971] is the classic article on the automatic theorem proving approach to program synthesis. [Buchanan and Luckham 1974] and [Buchanan 1974] are more recent, representing the current state of research. There is a great deal more literature in this area, but these are representative.

drawbacks. Each of the prerequisites has difficulties associated with it. Having a formal description of the input and output means that the user must have these expressed completely and expressed in the language of the theorem prover, usually first order predicate calculus[18]. Unfortunately, this is rarely an easy task. The specification that the user has is usually too general to be expressed this way. In our saving account example, the only real output is the current balance, the rest of the output is dialog with the user. However, most of the work of writing that program is in discovering the appropriate representations and relationships between data, and not in the transformation for the output. Also, the specification given by the user is often incomplete. It can be incomplete in two different senses. For a human programmer, the specifications tend to evolve as the implications become apparent while designing and testing the program. He might be willing to accept a small change in the specifications that will result in an improvement in the resulting programs. Secondly, incompleteness may exist because the user does not care about certain kinds of details or assumes a certain domain which should be able to supply appropriate defaults. The predicate calculus is just not flexible enough to handle this kind of variability conveniently.

Providing a description of the environment is a bigger problem. The description is treated as a constraint space into which the prover attempts to fit the program. If the description is not complete enough, there is a danger of admitting non-standard models, not consistent with the real world[19]. On the other hand, if too much detail is included in the environment description, the theorem prover will be slowed down.

---

[18]The system of Darlington ([Darlington 1973]) uses second order predicate calculus, permitting a more powerful descriptive capability, but a more difficult theorem proving task.

[19][Buchanan and Luckham 1974] p. 13 gives an example of how this might happen.

The power of the theorem prover is also a problem. Even the most recent systems require the user to provide such things as the configuration of the variables for the program, the invariants and tests for program loops[20], and plenty of assistance in ordering the goals and rules.

The most important problem with the theorem proving approach arises from the difference in goals between a theorem prover and a programmer. The goal of a theorem prover is to establish that the task can be done. To modify this goal to conform to the programmer's goal of finding a good program means modifying the control structure of the theorem prover to deduce the implications of its decisions and take suggestions. That is, while the theorem prover is working toward a proof, it is also making decisions that will affect the form of the program. Unfortunately, it is not clear how these decisions should be controlled.

The second approach to a program synthesizer is based on some observations about the way much of the human programming is done. It is exemplified by Hacker[21], and might be characterized the debugging approach. The task to be accomplished is stated as a goal for which Hacker is to produce a program. Hacker maintains a library of programs and methods for producing programs, from which the most applicable is selected. Hacker then tries running the code. If it fails because of a bug in the program relative to the new situation, Hacker debugs the code so that it accomplishes its purpose. If it fails because of missing code, Hacker recurses going back to the program library. Sussman calls this technique "evolutionary programming", since the code evolves as it is applied to new situations. This is often the way people program, especially while maintaining large programs.

---

[20] [Wegbreit 1974] presents some work in automating the synthesis of the loop predicates.

[21] [Sussman 1973], the program synthesis system of Sussman, exemplifying a theory of how skills might be acquired through trial, correction, and generalization.

The main difficulty with this approach for any kind of programming, but especially generating new programs, is the lack of global planning. There is no overview of the planning process in the system. The structure and generality of the programs are governed by the non-specific nature of the techniques used to solve the local problems, and not by directed use of the objectives of the program. This kind of a system is limited to situations where it is permissible for the programs to fail occasionally, because there is no assurance that a program will work for all of the cases on which it might be used. Since the programming situation to which this thesis is address includes information about the possible ways that a program might be used and does not include a debugging program to help out the program if it gets into difficulty, the strategy of Hacker is not appropriate.

Mycroft, the program monitor of Goldstein[22], although intended primarily as a system to debug simple programs written by people, is also built around the same evolutionary approach to programming. For debugging, it takes a set of facts about the intended result of the program (the programs draw pictures, so these are facts about shapes and relationships in the pictures) and the almost correct program. From these it decides what the plan for turning the facts into a program must have been. Then it determines why the program deviates from producing the intended result. To turn this system into a synthesis system, Goldstein suggests the following: start with a subset of the facts about the result, produce a plan to satisfy these facts, produce a procedure for this plan, continue in the same fashion for any sub-procedures, and then debug the resulting procedure to satisfy the remaining facts about the result. Producing the procedure for a plan is done in a linear fashion unless parts of the desired picture are

[22][Goldstein 1974], a program understanding and debugging system, emphasizing the use of facts about the intended result.

described in terms of a generic element, indicating a need for iteration (called round plans). This strategy for producing programs is "evolutionary" in the sense that the facts that were not taken into account on the first pass cause the program to evolve into the desired program.

The third approach is based on examples. There are a number of ways of using examples to produce a program. Green in [Green et. al. 1974] lists four: example input-output pairs, program traces, generic examples, and generic traces. The input-output pairs have received the most attention[23]. The approach is to use the input-output pairs to suggest the general input-output mapping and then produce a program that implements that mapping. Since the finite set of input-output pairs (in some cases only one) could be extended in an infinite number of ways, the system must have some notion of a natural extension. The system depends for its usefulness on how well this notion corresponds to the user's desires. The approaches of these systems vary from a heuristic set of rules to determine what was most probably intended to completely algorithmic abstraction of recurrence relations. The program trace approach[24] operates at a lower level, taking a trace or partial trace (i.e., the changes it makes to data structures, internal and external) of the program and produces an appropriate program structure that would have such a trace. This approach has similar kinds of problems with the naturalness of generalization. Generic examples and generic traces are compromises to still give the user the ability to specify by example while partially clearing up the ambiguity of the possible generalizations. These systems are really aimed at a different problem than the one addressed by this thesis. They have recognized that it is hard to specify what a program is to do. The programs they produce are in reality more like the

[23] [Summers 1975], [Summers 1976], [Shaw, et. al. 1975], and [Hardy 1975]

[24] exemplified primarily by [Biermann and Krishnaswamy 1974] as a means of synthesizing programs

specifications for programs. A more complete system would take the result of the example abstraction and expend more effort determining an efficient implementation. The Stanford group ([Green et. al. 1974]) has recognized this and uses the technique as one of the ways of specifying a program in their current work[25]. Even as a specification technique examples have drawbacks -- only for a limited domain of problems are the examples sufficiently revealing and the possible ambiguity of the generalizations require some sort of negotiation with the user to be sure of his intensions. Other automatic programming projects such as the one at ISI ([Balzer 1975]) have as their primary thrust the specification of a programming problem. Since that is not a major part of this thesis, it will not be discussed further.

The fourth approach is the refinement approach. This is the most closely akin to the approach of this thesis. The PSI project at Stanford[26] is the primary example of this approach. The method of producing programs is to refine the algorithm and data specifications until the result is a program in the target language. Because, as discussed in the previous section, refinement is not sufficient to organize the synthesis process, the PSI system consists of a number of parts, one of which does the refinement. There is a basic division in PSI of the design process into two parts. The first part, called the *acquisition group* consists of five modules that cooperate in building a high level general specification for the desired program called the *model*. In the process the modules are also responsible for the global consistency checks, recognition of patterns, and formulation of the basic algorithm. That is, any of the functions of a data model or argument model that involve non-local properties are handled by this group. The second part, the *synthesis group* consists of a coding expert that generates refinements to

[25] i.e., the PSI project, [Green 1976]

[26] [Green 1976], [Barstow 1977], [Barstow and Kant 1976], and [McCune 1977]

design and an efficiency expert that selects from the possible refinements on the basis of time-space estimates for the resulting program. This part works within the framework set up as the *model* without making changes to the basic structure. The coding expert operates from a knowledge base of production rules to apply to the design structure. It does not make use of any global properties of the design in applying those rules, implying that any coordination of brother goals (refinement of objectives in different parts of the program that have the potential for interacting) is only handled by the efficiency expert. It in turn operates by comparison of estimated time-space products.

The contrast between this organization and that of the programwriter is in the way functions that would be handled by models in the programwriter are handled. The *model* (detailed program specification) produces a boundary between higher levels of abstraction and lower levels. The functions of a domain model and the more global functions of data and argument models are restricted to the higher levels of abstraction. The functions of a target language model and the more local functions of data and argument models are restricted to the lower levels of abstraction. (I/O does not have enough importance in the domain to be mentioned. Because the domain is single programs, some of the functions of the data model are not needed.) The result is several restrictions on the system:

1) Any information about the domain must be applied at the higher levels of abstraction.

2) Any influences of the target language will not be felt above the lower levels of abstraction.

3) All of the global modifications must take place at the higher levels, although the only place the efficiency is considered is in the lower levels.

As a result, the level of the language for the *model* is critical in determining the kinds of

situations the system can recognize and handle. In more complicated domains where efficiency considerations have to be introduced at a higher level, domain information has to be available at a lower level, or other models introduce special restrictions on the form of the program, this rigid boundary may break down.

Another example of the refinement approach is Protosystem I[27]. In this system the domain of the problem has been limited to a particular kind of management information system. Within this domain the problem can be formalized to a much greater extent than in the general program synthesis problem. There is a great deal of understanding of the kinds of issues confronting the designer of a management information system and the techniques that are the most consistently useful. Within this framework a particular organization and set of techniques can be provided that will generate good solutions to a large class of problems. The refinements of a specification can be organized into a fixed set of phases with the desired information available when it is needed. The specification can be turned into a basic program model in terms of data set manipulations. The basic model can be globally optimized by aggregating computations and data sets. And, the result can be turned into programs. The whole structure and approach takes advantage of the knowledge about the domain. The system shows the advantage gained for a practical system by limiting the domain and utilizing what is known about programming in the domain to organize the system. The programwriter utilizes this philosophy to some extent in the data model by utilizing a particular flexible intermediate *data base* representation for the data it needs to store, once those needs have been discovered, rather than trying to go directly to some target language representation.

---

[27] [Ruth 1976-1], [Ruth 1976-2], and [Morgenstern 1976]

3.1 Other Aspects of the Problem

Besides the four approaches to the synthesis problem, there is a great deal of research that bears on different aspects of the problem from how programming is done by people to methodologies for making use of programming knowledge. A little of this will be mentioned here and more will be scattered throughout the thesis in the appropriate places.

Ruven Brooks in his thesis ([Brooks 1975]) examines the way that people actually do programming. His theory is that the programmer goes through three phases, perhaps backing up when difficulty strikes. The three phases are understanding, planning, and coding. He uses the protocols of programmers working on some sample programming problems to test this theory. The programming problems, which involve simple operations on sets of numbers, take a programmer about half an hour each to complete. He states that for the typical problem understanding took about 2 minutes, planning took about 5 minutes and coding took the rest of the time. Little or no evidence of search activity was seen in the planning stage, leading him to conclude that the planning was basically a matching operation. This is very different from the protocol of Naur[28] working on a more difficult problem which required a considerable amount of search and false starts. Ruven also states that the coding activity starts once the plan is generally sketched, even though some of the details are still unknown. The coding then seems to be a design task which fills in those details as needed.

In comparing Brook's approach to the programwriter's approach there are several observations that need to be made. Protocols disguise a lot of the low level

[28][Naur 1972], an informal protocol of the author attempting to write a program to solve the "eight queens problem"

processing that might be happening. The programmer reports the major observations and decisions without noting the low level checking and decision making that he may not even be aware of. This is further complicated by the fact that a programmer rapidly builds up shortcuts and "macros" that save effort. These shortcuts must at some point have been derived during the normal planning and coding activity. The programmer, because of limited short term memory, also needs to write down reminders after the design has reached a certain level of detail. All of these factors lead me to believe that much of the actual planning activity is hidden in the coding part of a protocol. In any case the protocols and theories about them are very useful in suggesting the organization and function of the target language model.

The work of Rich and Shrobe ([Rich and Shrobe 1976]) on a programming apprentice also suggests capabilities for the target language model. Their system, since it must respond to the constructs of the user, needs a great deal more knowledge about the consequences of each construct. The programwriter, as an example system, can get by only knowing the important (with respect to the programming domain) consequences of the constructs it uses. Eventually of course, the generalizations of the programwriter will have to have capabilities such as those proposed by Rich and Shrobe.

There is also a rich literature in the handling of planning activity. The work of Sacerdoti[29] provides a method of organizing multiple goals with conflicting requirements. The non-linear organization of his plan space is similar to the minimal constraint organization of the programwriter's program designs. The work of Waldinger and Manna[30] present some of the reasoning paths that turn some of the typical requirements of specifications into the constructs that handle them. The programwriter at the stage

---

[29] [Sacerdoti 1975-1] and [Sacerdoti 1975-2]

[30] [Manna and Waldinger 1974] and [Waldinger 1975]

presented by the thesis is not confronted with the problems solved by either of these sets of work, but again generalizations of the programwriter will have to have such capabilities.

Section 4 Organization of the Thesis

There are five more chapters in the thesis, each one serving a different function. Chapter two explains the structure of the programwriter. It explains the language used for representing the structures, the form and use of the structures themselves, the organization of the programwriter, the form of the specifications and the ultimate programs, and the basic operating routines of the programwriter. Chapter three is about the models used by the programwriter. It explains each one -- the basic view of the programs, the different modules that make it up, the way the modules work, and the parts it plays in the programming process. Chapter four is a set of four scenarios of the programwriter designing programs. The first scenario, the basic one, is also covered in an overview in the introductory section of the chapter. Chapter five presents a collection of topics concerning the more difficult questions the programwriter is called on to face in the scenarios. Chapter six contains the concluding remarks and recommendations for future research.

Chapter II
How to Build a Programwriter

The process of designing programs within this environment of refinement driven by models places certain demands on the structure of the programwriter. The introductory example required the use of several different modules and the selection and instantiation of several different structures just to bring one node to the point where it could be refined. To coordinate the use of these modules and provide the structures when they are needed, there must be a flexible control structure in the programwriter. The different modules are associated with the parts of the models and must be called when they are needed. There are static and dynamic information structures that must be handled in a consistent manner. And, there are analysis procedures that must maintain the current state of the design. The whole refinement process must be controlled by the programwriter to maintain an appropriate order among the various modules with jobs to perform.

This chapter covers the basic structure of the programwriter. The first section describes the environment in which the programs written will be executed. The environment provides certain services and conventions for the programs. It determines what can be counted on during the execution of the program and what must be provided by the programs themselves. The second section provides an overview of the programwriter. It explains the major parts and how they fit together. After the second section, the sections can be divided into three groups. Sections three, four, and five discuss the general issues of representation: the syntax of Owl-I, the kinds of relationships it can represent, the meaning of the expressions, and the specification language. The basic representation of concepts (the representational unit) in Owl-I is in

a "is-a" hierarchy. That is, the concepts are in a tree where each concept is a more specific kind of the concept above it. Because this places a constraint on the relationships that can exist between concepts (there is only one upward "is-a" link per concept), the hierarchy has been extended by having concepts that are *substantive characterizations* which can represent concepts that would not fit comfortably in a single place in the hierarchy. This, along with the ability to make other kinds of links between concepts, gives structure to what have been called "is-a" links in such systems as that of Fahlman ([Fahlman 1975]). The primary links are provided by the hierarchy. Secondary links are provided by substantive characterizations. And, other links are provided by other kinds of relations. The final section on representation (section five) discusses the handling of time. Time is one of the more difficult aspects of dealing with the writing of programs, because possible time relationships that will take place in the future must be anticipated and handled.

Sections six and seven deal with the knowledge base structures the programwriter uses in the design process. The static structures include *methods, intents, schemas,* and *definitions.* The *methods* are the basic plans for refinement. They correspond to the rules in a production system such as PSI ([Green 1976]) or the macros in a macro expansion system such as Hacker ([Sussman 1973]). The *intents* are structures to declare the basic properties of a piece of design structure (called a node) -- similar to PLANNER antecedent theorems ([Hewitt 1972]). They declare the properties of a node that would normally be required for other stages of the design process. The *schemas* are structures used to carry out more extensive kinds of planning activity. They provide the ability to make changes or additions at more than one node at a time. Refinements of data structures or modifications of the algorithm involve more than a single node, requiring a mechanism that will coordinate the various operations. The

modification of an algorithm in particular is a procedural operation with certain steps that must be performed. *Schemas* give the programwriter the ability to include such procedural operations in a natural way -- an ability lacking in production systems. The *definitions* provide the relationships that exist between concepts in the domain. They supplement the specifications by filling in any missing information.

The dynamic structure, the program design tree, is discussed in section seven. The primary feature of this structure is the completely open nature of the information. All of the past refinements are available to be analyzed or modified. This section also discusses the *goals* which constitute the pending needs of the design process. And, it discusses the *ideas* which are the pending suggestions for alternatives.

The final section discusses the routines that service the needs of the design modules. They provide the other modules with the ability to evaluate concepts in the current environment and determine the values of predicates. Because of the time considerations for a design environment, these routines must be able to deal with future possibilities.

Section 1 The Program Environment

The programwriter addresses the problem of a certain kind of programming situation. In this situation the programmer is confronted with specifications for an interacting set of programs. The specifications are complete in the sense that they are the only programs that need be considered in making decisions about the data structures or formats. They are not complete in the sense that information in the specifications must be supplemented by the programmer's knowledge of the programming terms and

the properties of the domain. Thus, the assumed domain is one in which the programmer has a fixed closed set of program specifications representing real world events. The programmer is expert in the knowledge about programming, and has reasonable familiarity with the events the programs represent through the domain model.

It is assumed that these programs will be executed starting at some time in the future. At that time there will be no pre-existing data. At that initial time it is possible to run a special program called INITIALIZE, which will set up any structures that might need to be initialized for the other programs. After that any of the programs can be run one at a time, subject to any constraints in the specifications. The programwriter must deal in terms of possible execution histories to answer questions about what is needed. The model of this execution history time is fairly simple. All times are expressed in terms of days, which are numbers starting from some arbitrary day for the initial time. They are given decimal values representing the time of day so that different events may have times of execution in the same day. Thus the times of execution of three programs all on the same day might be 7.1, 7.2, and 7.3. A constant period of time such as a month is represented as a unit of TIME, 1-MONTH, equal to 30 in arithmetic calculations. The execution of a program takes place at a specific time, which can be determined by a call in the program to the Lisp primitive TODAY. During the execution of a program, the value returned by TODAY will remain constant. Thus, the program execution events take place at points in time and these points of time can be used to identify a particular run of a program. With this time model, the handling of time in the programs is possible with the simple arithmetic predicates and operations.

There are also a cast of characters in this programming environment. First, there is the person who requests that the program be written. This person, the

requester is not needed after the specification is received. In a generalized system, there would be interaction with the requester to answer questions, but the assumption is made that all of the answers are already in the knowledge base. This causes no loss of thrust for the main points of the thesis. The person the programwriter will have to consider most is the one running the programs -- the user. In reality, when a deposit is made at a bank, there is the depositor, the teller, and possibly someone running the program. For simplicity, it is sufficient to ignore the depositor and the teller except as possible parts of data (i.e., the program may be required to keep track of the teller's name for each transaction) and assume the "user" is the one interacting with the computer.

The ultimate objective of the programwriter is to produce programs in Lisp that implement the specifications. These programs will take no arguments, leaving the securing of any needed information to I/O exchanges. The whole program history takes place in a single Lisp environment. This means that global variables and property lists are considered to be permanent storage. Indeed, the programwriter does not know about external storage. This is a simplifying assumption, but justifiable since external storage presents the same general problems except that there are many installation dependent details. The details of handling the Lisp constructs are considered representative of the kinds of problems involved in external storage.

Section 2 Overview of the Programwriter

The major parts of the programwriter are illustrated in figure 5.

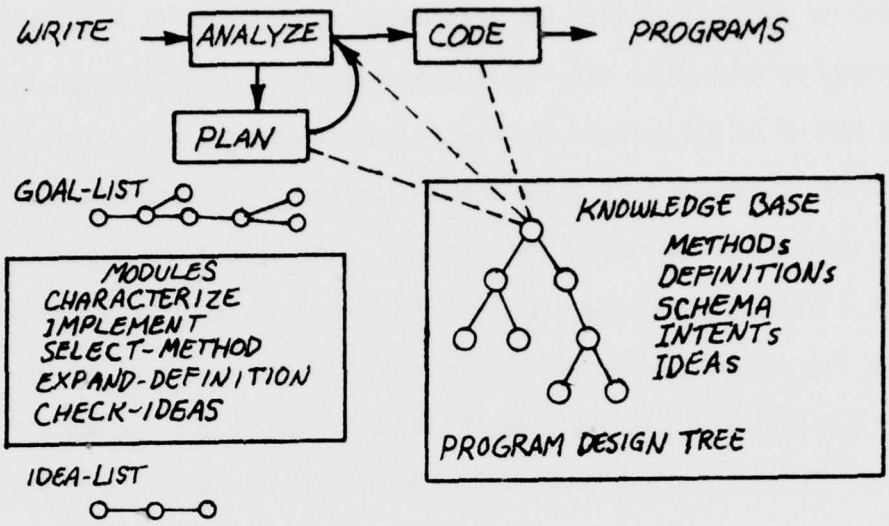

Figure 5. Programwriter overview

The *main loop*, consisting of the *analysis*, *planning* and *coding* routines, drives the programming process. The programwriter is started by calling the Owl-1 procedure WRITE with the desired specification. This procedure calls the analysis and planning routines alternately until the analysis routine determines that the design is complete. Then WRITE calls the coding routine to produce the final Lisp program. The information structures are all implemented as *concepts* in the Owl-1 language and located in a single *knowledge base*, allowing links between any of the structures. The specification given to WRITE will be used as the beginning of the *program design tree*. The program design tree is the structure in the knowledge base representing the state of the design during the processing. Each of the new decisions made by the analysis and planning routines is added to it. It is this tree that contains the nodes and other information produced during design. In the short example in the introduction, the ACCEPT-DEPOSIT node, the ASK node, the BECOME node, and the ASSERT node are all parts of this tree. The analysis and planning routines call the modules, which utilize the other kinds of information in the

knowledge base. Figure 6 shows in more detail the relationship between the analyze and plan loop, the modules, and the information structures. The arrows indicate calling relationships and the dashed lines indicate data use. All of the modules also use the program design tree. Also, any of the modules may call on the service routines EVALUATE, WHETHER, and SUITABLE-REPRESENTATION for evaluation, predicate determination, and pattern matching respectively.

The analysis routine is the part of the loop that provides most of the raw information and goals for the planning routine to work on. The purpose of analysis is to explore new nodes, collecting any suggestions for improvement (called *ideas*) they may point to and determining locally how they fit in with the rest of the program design tree. The first time analysis is called it goes through the specification and operates on each specification for a program separately as a new call. When analysis is called after the first time it goes through the program structure that has been built up and picks out the new calls in the algorithm refinement added by the last planning step. It then analyzes each of these calls plus any specific requests for analysis made by other modules.

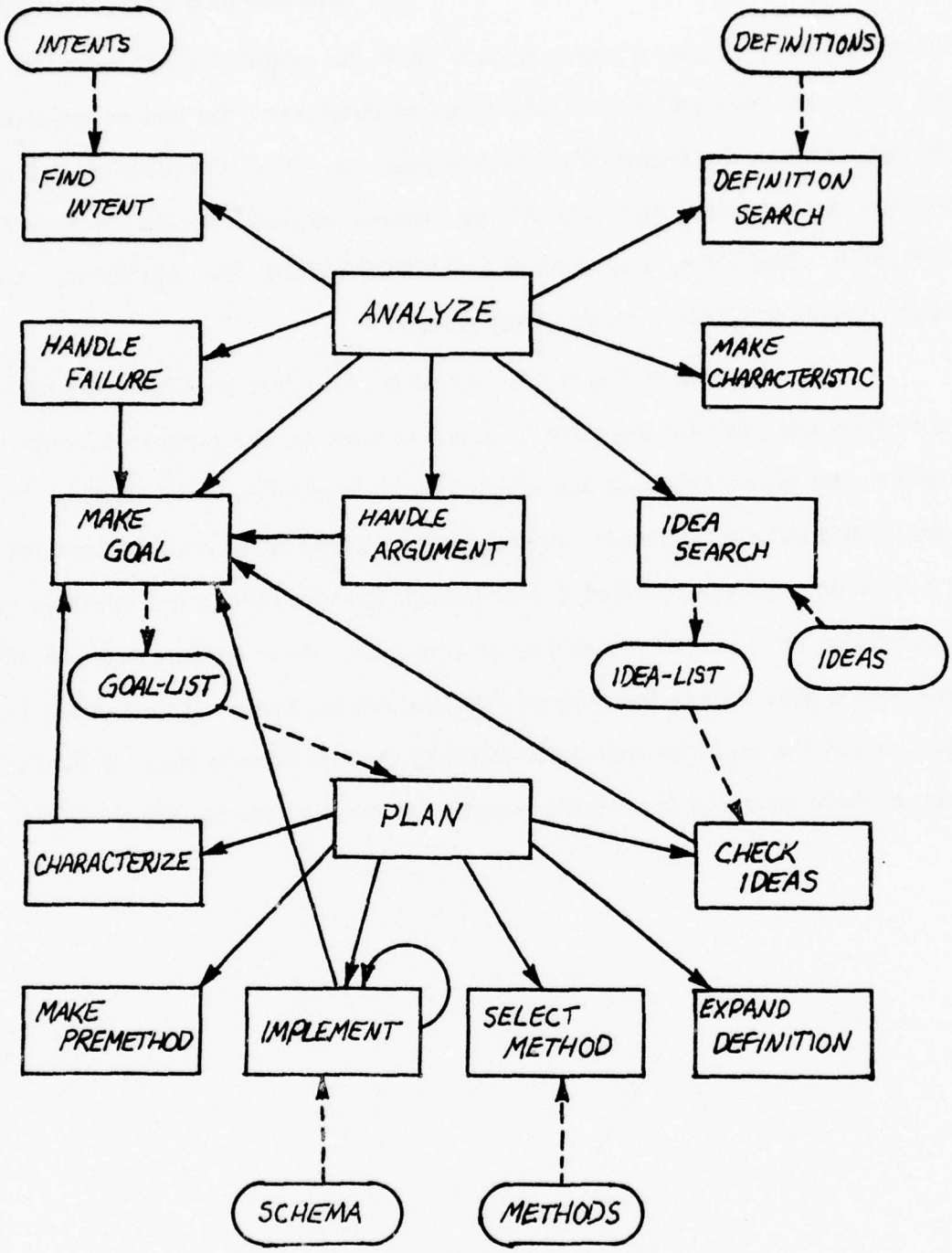

Figure 6. Analyze and Plan

```
ANALYZE (for a call that is an ACTIVITY)
  1)Evaluate call
  2)Make new node
  3)Find an INTENT
  4)Attach INTENT to node
        Make characteristics
        Make goals
        Handle arguments
        Handle failure
  5)Search for IDEAs
```

Figure 7. Analyze Algorithm

The first step in analyzing a call is to evaluate it. This fills in any variables with their current values. The next step depends on the kind of call. If it is an operation (called an activity), the evaluated call is tied into the program design tree as a new node in the appropriate place. The next step is to call the FIND-INTENT module, which finds an appropriate INTENT from those in the knowledge base. The INTENT provides a package of declarations to be made about the node. To handle those declarations the analyze routine will call MAKE-CHARACTERISTIC to add any indicated characteristics, and call MAKE-GOAL to add any new goals to the GOAL-LIST that the planner should worry about. If the INTENT declares that the node takes arguments, the analysis routine calls HANDLE-ARGUMENT with each of the arguments. This will check to see that there is a node that can produce the argument or call MAKE-GOAL if a node needs to be added. If the operation is one where failure is possible, analysis calls HANDLE-FAILURE. This routine also may cause more goals to be created if code must be included to handle a failure.

If the evaluated call is a BECOME (an indication of a state change), the analysis routine checks the node above to see if there are any calls stored there that belong in place of the BECOME. If there are, these are attached to the node as new

calls for analyze to handle as above. Many times the BECOME is used for information during the design, meaning that it does not require any nodes. In that case, the evaluated BECOME is just used as information for any modules exploring the tree.

If the evaluated call is a control structure (IF-THEN or IF-FAIL), the predicate must be processed by WHETHER and then the appropriate parts can be turned into nodes in the same manner.

The final step of evaluate for all of the nodes is to call IDEA-SEARCH. This routine uses the node to search for IDEAs that might lead to improvements in the program. Any IDEAs that it finds will be included in the IDEA-LIST for later use by the CHECK-IDEAS module.

The result of all of the analysis is that the essential features and requirements of each new or newly influenced call in the program design tree are made available to the rest of the system.

2.1 Planning

The planning routine is simpler than the analysis routine, since it is mainly a dispatch mechanism to call other modules. It has the responsibility for carrying out the design of the programs, while the analysis routine serves in an information gathering role. Planning operates from the GOAL-LIST, built mostly by the analysis routine, but also added to by some of the other modules. Planning selects a goal and finds a module to handle the goal. Selection of a goal utilizes partial order constraints in the GOAL-LIST, and the priorities of the goal classes. Once a goal has been selected, the appropriate module is called by dispatching on the form of the goal. If the module succeeds in

handling the goal, it returns SUCCESS. The goal is then marked as complete, removed from the GOAL-LIST and the planning step returns. The module can fail for two reasons, either the goal had already been completed or it could not be done. If in the handling of a goal, the module discovers a need for some other goal to be accomplished before the current one can be completed, it calls MAKE-GOAL with the offending goal and constrains the GOAL-LIST. When a module fails, the planning routine chooses.

The modules called by the planning module correspond to the general goal classes that involve planning and include MAKE-PREMETHOD, EXPAND-DEFINITION, SELECT-METHOD, CHARACTERIZE, CHECK-IDEAS, and IMPLEMENT. There are also some special purpose versions of these that match more specific goals. Their general function is as follows:

1) MAKE-PREMETHOD puts a new call into the program design tree. Actually, it creates a specialized method that includes the new call in the desired place.

2) EXPAND-DEFINITION takes the general form of a definition and turns it into the specific instance most appropriate for the specification -- basically an evaluation process.

3) SELECT-METHOD searches for a refinement appropriate to the node. The refinements it uses are the METHODs in the knowledge base.

4) CHARACTERIZE determines the important properties of data, such as what parts are needed, when it is defined, what its sources are, where it is used, whether it can change, and so forth. If it discovers any missing operations it will call MAKE-GOAL to have the problem handled.

5) CHECK-IDEAS tries out the IDEAs for improvement to see if any of them will be beneficial to the program. It uses the IDEA-LIST built by analysis. It will call MAKE-GOAL to set up any implementations required by the IDEAs.

6) IMPLEMENT is an interpreter to carry out SCHEMA. It takes a pattern for a SCHEMA, finds one that matches from the knowledge base, and carries out the steps. The steps may cause more goals, produce new steps, or call IMPLEMENT again.

Section 3 Representation in Owl-I

The program design tree, the knowledge base, the goals and ideas, and the main loop are all represented in a knowledge representation language called Owl-I. Owl-I is an early version of Owl[1], a knowledge representation system under development by the Knowledge Based Systems Group at the MIT Laboratory for Computer Science. Owl consists of: LMS (Linguistic Memory System), the data base system [2] which handles the reading, storing, and printing of *concepts*; a theory of English grammar[3]; a *world* of concepts taxonomically structured in LMS; and an interpreter for executing procedural plans composed of the concepts[4]. Owl is designed to facilitate the representation and use of the kinds of knowledge involved in verbal reasoning. It is modeled on English to take advantage of the structure inherent in natural language and to simplify translation between the English description of knowledge and the Owl representation. Other projects that have been build on the foundation of Owl include an English language dialogue system ([Brown 1977]), a digitalis therapy advisor capable of explaining what it is doing ([Swartout 1977]), and a system that assists the user in designing a hierarchical planning and control system for a procurement firm ([Bosyj 1976]).

The Owl-I system used by the programwriter utilizes the LMS data base for the organization of its concepts. The English language basis is utilized in suggesting the structures for concepts and in simplifying the interpretation of specification, but parsing and generating natural language (except to the small extent needed by the I/O model) are not part of the thesis. The *world* of concepts used by the programwriter is smaller

---

[1] The basic ideas and representational scheme of Owl are discussed in [Szolovitz et. al. 1977]

[2] designed and programmed by L. Hawkinson [Hawkinson 1975]

[3] under development by W. A. Martin ([Martin 1977])

[4] [Sunguroff 1976] and [Long 1975] discuss the implementation and approach of the interpreter.

in size, but more complex in links between concepts than the ones used in some of the other applications of Owl. It has the same basic structure with fewer general language concepts, although it has been extended to include more concepts pertinent to programming. The interpreter, which has been modified extensively, is used to run the main loop and execute some of the special structures (the schema, which will be discussed later). Mainly however, the modules of the interpreter for evaluation, question answering, and pattern matching are used to assist the modules of the programwriter. Because of the special needs in designing programs these too have been extended to handle, among other things, anticipation of future possibilities.

The whole Owl-I (and Owl) system is implemented in Lisp and is intended to handle both procedures written in Owl and in Lisp. Most of the modules of the programwriter are written in Lisp, but all of the information structures are in Owl-I. It is designed to make it possible for Lisp procedures to use Owl-I or Owl-I procedures to call on Lisp procedures. Because of the internal circular structure of Owl-I, it is necessary to use the Owl-I reader and printer to handle concepts.

This representation language provides the programwriter with a number of features it needs. From the example it is apparent that the representation must provide for relations, format for static structures, the capability of building new structures, and an organization in which it is reasonable to search for structures. In fact as will become clear later, there are also other requirements such as sets represented by a pattern of their elements, the inheritance of properties from more general concepts, and the ability to return to previous values of relations. Owl-I provides these capabilities through the highly connected data base structure and the routines that maintain and extract meaning from the structures.

The basic unit of representation is the *concept*. A concept consists of three parts; a genus, a specializer, and a reference list. The genus and specializer uniquely identify the concept and provide its primary relationship to other concepts. The genus is itself a concept, joining the whole data base into a taxonomic tree making partitioning of concepts by type and inheritance of properties from higher level concepts possible. The specializer may be a concept or a symbol (marked with quotes, as in "BALANCE"), and provides a classifier link between concepts. All concepts can be represented as (*genus specializer*). For example, the concept (BALANCE ACCOUNT) is a kind of BALANCE for ACCOUNTs. It has the concept BALANCE as genus and the concept ACCOUNT as specializer. The most general kind of link to other concepts is provided by the reference list of the concept. The reference list has on it the other concepts associated with the concept, including the value (if meaningful), characteristics, properties, and the uses of the concept as genus, specializer, or value of other concepts. With this structure a very rich knowledge base can be built.

A few pointers on Owl-I syntax[5] are necessary to understand the code in the rest of the paper. A concept is represented by a label that names it (a label is defined with "=") or if one does not exist, by the specializer if it is a symbol (any Lisp symbols inside Owl-I expressions are in quotation marks), or by representing the genus and specializer within parentheses. Thus:

```
[MY-BALANCE=(BALANCE (ACCOUNT ME))] is MY-BALANCE
[(AMOUNT "BALANCE")] is BALANCE
(BALANCE ACCOUNT)
```

In the scenario there will also be concepts with labels starting with "*". These are local labels valid within the particular section of Owl-I code where it occurs, meaning that it is

[5] The representation for this paper has been simplified to avoid some of the more confusing details. Therefore, it does not conform to all of the conventions of "standard" Owl.

possible to use the same label again later for some other concept. Sections of Owl-I code are enclosed in square brackets. Such a section is called a *complex*. Inside there are one or more concepts, which may include more complexes. A complex has a procedural interpretation in which the printed form stands for the first concept inside the square brackets, with each of the other concepts in the complex placed on the reference of the first concept as a side effect. Thus, [A B] is A with B on its reference list. Inside the complex, the first concept can be referred to with a colon. Actually, a concept may be inside several levels of complex and the first concept at any level may be referred to with a quantity of colons equal to the number of the level. That is, the outermost complex's first concept is ":" and so forth. Thus in [A [B C: D::]], (C A) and (D B) are on the reference list of B and B is on the reference list of A. Consider one of the METHODs used in the refinement process:

```
[(METHOD (ACCEPT DATUM))
 OBJECT: <- DATUM:
 STEPS: *ASK=(ASK DATUM:),
        (BECOME (DATUM: THE) <- (RESULT *ASK))]
```

The concept that this complex stands for is (METHOD (ACCEPT DATUM)). On the reference list of the concept are OBJECT: and STEPS:, which in turn have concepts on their reference lists. It says the object of the METHOD is a datum and the steps are to ask for the datum then the result of asking becomes the datum. The *ASK is a local label, valid inside the METHOD, standing for (ASK DATUM:). The "<-" indicates value. Thus, the value of OBJECT: is DATUM:. DATUM: is actually (DATUM (METHOD (ACCEPT DATUM))) meaning "datum of the method of accepting a datum". The DATUM:, because it is specialized by the METHOD, is a variable and will be bound to some actual datum by the programwriter when it uses the METHOD. OBJECT: and STEPS: are also specialized by the METHOD but they are recognized parts of the METHOD. In the STEPS, the comma indicates the order of the steps and is shorthand for a THEN concept connecting the two

steps. That is, [A, B] is [(THEN A) <- B]. Thus, the *ASK step is to take place before the (BECOME ...) step. The "<-" inside of the parentheses of (BECOME X <- Y) indicates that X has the value Y in the context of the concept, rather than globally. Thus, the concept is a statement about Y being the value of X, and is used in this case to say that Y will *become* the value of X when the concept is executed. There are also a few other constructions that will appear in METHODs and other knowledge base structures, such as (IF-THEN X Y) used to indicate that the step Y is conditional on the predicate X; and (AND A B C) used to indicate groupings of various kinds. These are really not violations of the one genus one specializer rule, internally the concepts conform while having a configuration recognizable by the modules, reader, and printer.

3.1 The Capabilities of the Representation

This section will discuss the relationships of concepts to one another and the basic functions that concepts can have. As mentioned, Owl-I provides the programwriter with a number of the capabilities that it needs. One feature of the knowledge base is the hierarchical organization of the concepts. Every concept has a concept as its genus with one concept, SUMMUM-GENUS, serving as the top concept and having itself as genus. This "tree" of concepts, relative to the genus, makes it easy to determine if a concept is of a certain kind, to inherit properties from more general concepts, to dispatch to routines specific to a particular class of concepts, and to form patterns for desired concepts. Typical world taxonomies are presented in [Szolovitz et. al. 1977] and [Martin 1977].

Determining if a concept is along the genus path of another is needed to limit the amount of knowledge different modules have about the whole system. For example,

if a module only needs to deal at the level of ACTIVITYs[6], it can determine that a particular concept is a kind of ACTIVITY (called a *specialization* of ACTIVITY) and deal with it on that basis without knowing all of the ACTIVITYs. Different models and hence different modules have varying needs for detail of information about particular concepts, making this an important feature. Having a tree structure also makes possible the inheritance of properties from more general concepts. This is used extensively by the domain model to represent default values, general properties, and to provide definitions. The tree also provides a ready dispatch mechanism for many of the modules which are specialized to certain kinds of concepts. Any such procedure can be placed on the reference list of the appropriate concept and can then be accessed through the tree. For example, when it is necessary to select a method for one of the low level I/O functions, the goal will be something like (SELECT (METHOD (PRINT (STATEMENT ...)))). There is a special module that incorporates the I/O model's ideas about how to select such a method on the reference list of the concept (MODULE (SELECT (METHOD PRINT))). For the goal to (SELECT (METHOD (ASK ...))) used to find one of the methods for accepting a deposit, the general module on (MODULE (SELECT METHOD)) is used.

A hierarchy is not always sufficient to categorize the concepts when there will be different viewpoints represented. A concept may need to stand for concepts in more than one part of the concept tree. For example, the birds, reptiles, mammals categorization is not convenient for categorizing night animals and day animals. The programwriter has the same problem, making necessary to provide a facility to structure alternate views that may not conform with the primary hierarchy. Such alternate views are in the knowledge base as specializations of SUBSTANTIVE-CHARACTERIZATION and

[6]All of the verb-like concepts such as DEPOSIT, DO, PRINT, and ASSERT are ACTIVITYs -- a difference from later versions of Owl.

include such concepts as DATUM, TOTAL and LISP-ENTITY. These concepts must have DEFINITIONs in the knowledge base to explain what kinds of concepts in the primary hierarchy might match them and what properties they must have. SUBSTANTIVE-CHARACTERIZATIONs can be used just like other concepts, because the pattern matching and evaluation modules have special checks for them.

Concepts are used in several different ways by the programwriter including as generics, as patterns, and as specifics. The generic concepts represent a class of concepts, such as BALANCE, RETRIEVE, or BANK. These are not specific things, rather they represent the idea of *balance* or *retrieve* or *bank*. Generic concepts provide for the taxonomic structuring of the concepts and provide nodes to hold the properties applicable to all of the specific concepts that are specializations. That is, specific concepts inherit properties from the generic concepts on their genus path. Patterns are similar to generics, in fact a generic can be used as a pattern. They are used to act as a filter to accept or reject concepts for matching. Patterns can have properties required for matching, or be tied to other patterns specifying interrelationships that must exist between candidates for matching. Patterns are used to define sets, provide the requirements of a variable, and refer to desired parts of the program structure. Patterns can be represented either as generics, if their properties exactly match that of the generic, or as a unique specialization of the appropriate generic concept. Such a specialization is represented by the generic followed by a "#" and a number. For example, a pattern for blue cars might be [CAR#1 BLUE], which would match all specific cars that are also blue. The feature of this kind of specialization is that the searching routines know about it and can find such a concept in the tree. For example, if cars were categorized by make, the only place that information about blue cars could be put that would be inherited by blue Fords is under a concept like [CAR#1 BLUE].

Specific concepts represent actual things such as MY-ACCOUNT or a particular node in the program design tree. Specifics behave in ways analogous to the way the actual thing might behave. MY-ACCOUNT can have whatever properties are appropriate to accounts, and they can vary over time in the same way as an actual account. In cases such as data structures or program nodes the specific concepts can be created, just as real data structures are created. In this domain there are actually very few specific concepts that can be named at the outset of the programming. Instead, the program nodes and data structures are created during the design. Also, many concepts will be specific when programs are executed and must be represented as specific but unknown in the mean time. These are called dummy concepts[7]. They represent some as yet undetermined concept and provide a place to collect properties discovered about the concept. The way dummy concepts are represented is with a "*" and a number following the concept. For example, in the specification for one of the programs in chapter four, the account is ACCOUNT*1[8]. That means the ACCOUNT that the user wants handled. It is a specific account in any execution of the program, but not only does the programwriter not yet know which one it is, it must make the program work for any such specific account. Specific concepts can also be created for such purposes as representing a data base or other structure designed by the programwriter. This is accomplished by creating a dummy from the structures indicating what is desired and characterizing it as actual when it "becomes" into existence. It is occasionally necessary to use a dummy to represent a pattern. To do this, the pattern is given the characteristic PATTERN on the reference list. One important use of specifics is the program nodes. These are created from the steps of the Owl-I procedures used in refinement to represent the node and store the information about it. Each one is unique.

---

[7]These correspond to Brown's carriers [Brown 1977], Sussman's formal objects [Sussman 1973] and Hewitt's anonymous identifiers [Hewitt 1972].

[8]The "*1" actually stands for a more complex structure in which the concept is connected to a METHOD and an EVENT from which the concept can inherit properties, but that can be ignored as the properties will be included with the concept.

3.2 Relations

The knowledge base also gives the ability to form relations between concepts. Any time a concept is on the reference list of another, a relation is formed. The most basic relation is CHARACTERISTIC. For example, [MY-CAR BLUE] means that MY-CAR has the characteristic BLUE. These are used throughout the programwriter, but particularly by the data model. When the data model is *characterizing*, it is adding the CHARACTERISTICs it discovers to the data concepts. The most obvious relation is *value*. The value has a special place on the reference list to make it faster to use. Occasionally there are multiple values for a concept. This is represented by an AND of the values. Besides being able to have and assign values, it is necessary to talk about values. This is accomplished by using VALUE specialized by the concept. This concept can have on it any characteristics of the value. Besides the value, there are other named properties that a concept can have, depending on the concept, represented by specializing the property by the concept, as in (HEIGHT JOHN). The value of the property is indicated as [(HEIGHT JOHN) <- (FEET 6)]. There is a subset of the properties of DATUMs that act as the information bearing parts, and are therefore important for several of the models to distinguish from other properties. These are given the SUBSTANTIVE-CHARACTERIZATION PROPERTY.

3.3 Sets

The programwriter also deals with sets of objects. Sets are represented in the programwriter as the concept SET specialized by a pattern describing the elements of the set. For example, the set of all blue cars is (SET [CAR#1 BLUE]). Sets can also

have explicit elements, represented by giving the SET concept a value that is a list of the elements. Such a list is represented as an AND, as in [(SET LETTER) <-(AND A B C)]. Most of the SETs needed in the scenarios actually represent portions of sequences of data produced by programs. That is, they are used in the design of the program to stand for the sets of data that will exist when the program is executed. To understand the interpretation the programwriter must give to these concepts, it is necessary to consider the time frame for which they are used. The assumed time for which the evaluation of these SETs takes place is some time during the sequence of executions of the programs. The concept used in the design process to represent the set of A-DEPOSITs that will be encountered by a program during execution is (SET A-DEPOSIT). That is, A-DEPOSIT used as a pattern stands for any such data that has been created up to the point of execution and the set is all data that match the pattern. Sometimes only a certain property of the elements is required as in (SET (AMOUNT A-DEPOSIT)), which is the set of the amounts of the existing deposits. The set does not have to explicitly exist to be used. These sets will be used to describe the possible circumstances that might exist when the program is called. Restrictions can be placed on the set by restricting the pattern, as in (SET [A-DEPOSIT#1 ACCOUNT:: <- ACCOUNT-1]]), which is the set of existing deposits to account 1. Restrictions can also involve time, as in (SET [A-DEPOSIT#1 [ETIME:: (SINCE EVENT-1)]]), which limits the set to only those occuring since some event.

Given a set, it is useful to get its elements and find out its size. In particular, the Owl-I procedures need a syntax to reference the elements. The representation of the generic element is (ELEMENT (SET ...)). This will be encountered again in dealing with iteration. There are also two kinds of size that may be of interest. The most obvious is the number of elements in a set at a particular time. This is (COUNT (SET ...)).

The other is how its size varies over time. This will become important when considering the efficiency issues of the programming, when special terminology will be introduced to compare such "sizes".

Section 4 The Specification Language

The specification of a request to the programwriter is in the form of a call to the Owl procedure WRITE. WRITE takes as object a SET of activity calls to be turned into programs. These activity calls may have restrictions or properties to indicate the requirements for arguments and the use of the procedure. For example, the specifications for a savings account system with corrections (the specification for the third scenario of chapter four) is as follows:

```
[(WRITE [(SET (ACTIVITY "SAVINGS-ACCOUNT-3")) <-
        (AND (STATE (BALANCE ACCOUNT*2))
             (ACCEPT A-DEPOSIT*2)
             [(CORRECT *DEP=A-DEPOSIT*3)
              REQUIRE::: <-
              (> (ETIME *DEP)(DIFFERENCE (ETIME ACT:::) 1-MONTH))]
             (ACCEPT A-WITHDRAWAL*2)
             [(CORRECT *WIT=A-WITHDRAWAL*3)
              REQUIRE::: <-
              (> (ETIME *WIT)
                 (DIFFERENCE (ETIME ACT:::) 1-MONTH))])])]
```

The set of activities has a value which is the AND containing all of the calls. It must be a set with explicit elements because the programwriter is only prepared to deal with a fixed set of programs. Only by assuming that the programwriter already knows the whole set of requirements can it determine what code and data are really needed. This is not to say that the programwriter will not find reason to produce a set of programs different than what is requested. Besides finding it necessary to add INITIALIZE and MAINTENANCE programs, it may not be necessary to have some of the ACCEPT programs if the information they could produce is not needed.

The programwriter is prepared to handle three basic requests for programs. These are ACCEPT, CORRECT, and STATE. ACCEPT is a request to acquire the information corresponding to a real world event that has just taken place and do with it whatever is appropriate. In (ACCEPT A-DEPOSIT*2) the real world event is A-DEPOSIT*2[9]. An A-DEPOSIT takes place at a time, called the ETIME, which is the same as the time it is being recorded by the execution of (ACCEPT A-DEPOSIT*2). The concept ACCEPT carries along with it the fact that the ETIME of its specializer is the same as its own time of execution (its ETIME). That fact is used in a number of ways, but in particular it is not necessary for an ACCEPT program to ask the user for the ETIME of its specializer, because it can just execute the Lisp primitive TODAY. The concept ACCEPT does not imply anything about what should be done with its specializer. That must be inferred by the programwriter. CORRECT also acquires information corresponding to a real world event. The difference is that the real world event is required to have taken place in the past. Thus, the ETIME of the specializer must be less than the ETIME of the program execution. It is not necessary that the earlier event have already been received by an ACCEPT, but that is possible. The third request is to STATE something, in this case (BALANCE ACCOUNT*2). There is no information explaining how the balance is to be determined, or the proper way of telling it to the user, only that it must be done.

Requests can have two types of modifier: a REQUIRE property on the call as in the example, or a KNOW property. The difference between the two properties is who bears the responsibility for the assertions they hold. The REQUIRE property only applies to ACCEPT and CORRECT. It says that the program should indicate that an error has

[9]The reasons for this representation will be discussed in the domain model in the next chapter. For now it just refers to a deposit made by the user.

taken place if the property is not true. That is, the program is responsible for checking that the predicates are true before accepting any data and incorporating it into whatever information structures the program uses. In the specification above, both of the CORRECT programs have requirements that the ETIME of the data be more recent than a month prior to the execution of the program. There is also a requirement that is a part of the INTENT for CORRECT that the ETIME of the data be earlier than the execution time of the CORRECT program. Only if these two predicates are found true by the program (the ETIME is within the prior month) can the correction be completed. The KNOW property gives predicates that must be true at the time of the execution. The program does not need to verify the fact, but the programwriter can make use of it in making decisions. This might be used in restricting the time of execution of a program or used to state facts known to be true of the concepts but not stated in the domain model. The purpose of such a property is to make it unnecessary for the programwriter to generate tests for conditions that will not exist. For example the information about an ACCEPT program having the same time of execution as the real world event it represents is provided by a KNOW in the intent for ACCEPT.

These three request types provide enough power to represent the kinds of specifications the programwriter is designed to handle. Beyond this the programwriter can handle arithmetic expressions and predicates. The arithmetic expressions include PLUS (a diad for simplicity), DIFFERENCE, SUM (to handle a set of numbers), and COUNT (to give the number of elements of a set). More could be included but these are sufficient for the examples. The predicates include the standard numeric comparisons plus AND, NOT, and EXIST. EXIST takes a pattern and determines whether any such actual entities exist. This is used for testing whether a certain kind of data has been accepted and whether a certain type of program structure exists. Again, these are

sufficient for the examples and representative of the kinds of problems predicates engender.

Section 5 Time

An important part of the relationships that need to be represented in a programming environment involve the time of occurrence of events. There are even several different kinds of time that need to be represented. The most obvious kind of time is the absolute time provided during the running of the programs for the times of execution and explained above in the section on the program environment. However, to design programs it is necessary to be able to represent the potential relationships between these times.

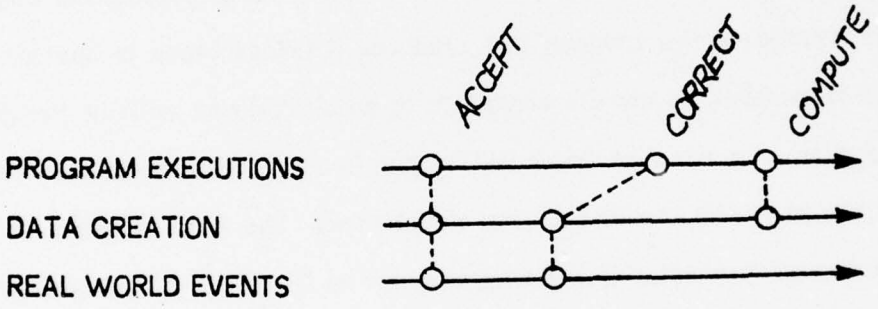

Figure 8. Time sequence of events

During the execution history there are three different sequences of events taking place: the execution of programs, the creation of data, and the real world events. And, there are rules to make the correspondences between the sequences. The easiest to keep track of is the execution history of the programs, because the times of execution are always available in the program by using TODAY. On the other hand, the programs and data stand for real world events, such as the depositing of money in the

bank or a user requesting the current balance. What the actual relationships are that exist between these real world events determines how time should ultimately be interpreted in the domain. Between the real world events and the program executions is the history of the data used by the programs. Data may come into existence as a result of a real world event or as the result of a program execution, depending on the type of data. In the case of ACCEPT these three kinds of events coincide. The time of execution of the actual event is known to be the time of execution of the program and also the time that the data representing the event comes into existence. With CORRECT, the time of the data and the real world event are still the same, but the time of the program execution is sometime after them. Anytime there is data that represents a real world event, the time the data comes into existence is the same as the time of the real world event. Data that is the result of a computation, such as producing the balance of an account, is related to the program that produces it unless there is specific notice otherwise. (It is possible to make a correction to an old balance without bringing the balance up to date -- a situation which will happen in one of the scenarios. Then the time of the balance is still the time of the old balance.) The time concepts for all of these situations are represented in the programwriter as the ETIME of the concept (time of execution). To determine the ETIME for data it is necessary to determine whether it has the time of the program that produced it or the time of some real world event. As a result, the ETIME of the data may be available from TODAY, it may be retrievable, or it may be necessary to ask the user.

Implicit in this discussion of the time of execution histories is the ability to talk about executions of a program or individual events. This is done using the concept ACT. For example, (ACT (ACCEPT A-DEPOSIT*1)) represents the execution of (ACCEPT A-DEPOSIT*1). In fact, A-DEPOSIT is a specialization of ACT, as it is an abstraction of

the real world act of depositing money, but that will be discussed further in the next chapter. The concept formed with ACT is still a generic, representing the class of such events. Specific instances can be formed by specialization, the use of elements and sets, or any of the ways specifics are normally used. In situations where only one such ACT would be refered to, the concept is taken to mean the most recent such instance. This occurs in specification of sources or the specification of a time period. The normal way to specify a time period is with SINCE. (SINCE (ACT (ACCEPT A-DEPOSIT*1))) specifies the time period from the last execution of (ACCEPT A-DEPOSIT*1) to the time of execution of the program in which the concept is located (call it the current time).

One other time concept deserves special mention -- INITIAL. It refers to a relative time, relative to some set of events. That is, INITIAL refers to the time at the beginning of the set of events under consideration. Generally, that time is at the beginning of the execution history, but it does not need to be. As will be seen in the scenarios, the initial balance when keeping subtotals of the balance is the last balance computed. An interpretation of INITIAL simpler than maintaining some context of events has been sufficient for the purposes of the programwriter. INITIAL is used as a specializer for data concepts. The value of the resulting specialization is used as a default value for the item itself. That way the datum specialized by INITIAL and the datum itself can be treated as separate quantities with different implementations if need be or the same quantity at different times if that is called for.

Section 6 Knowledge Base Structures

The building blocks used by the programwriter to form the design of the programs are structures in the knowledge base, such as METHODs, INTENTs, SCHEMAs

and DEFINITIONs[10]. Each of these is represented as a structure of properties on a single concept whose genus is the type of the structure. The single concept serves as a pattern declaring the purpose of the structure and the set of properties on the concept conveys the intended information. All of these structures are used by matching the specific problem to the structure of the desired type that bests fits the problem. Hence, they must have patterns that specify in what situations they apply. Another requirement for representing the structures is a mechanism to specify variables, making possible the binding of parts of the situation to the parts of the structure.

6.1 Methods

Most of the METHODs used by the programwriter have one or two steps, like the METHOD for ACCEPT, however there are a few that need more. Take a look at a more complicated METHOD:

```
[(METHOD (FIXUP AMOUNT))
 OBJECT: <- AMOUNT:1
 FOR: <- AMOUNT:2
 STEPS: [*RETR=(RETRIEVE AMOUNT:2)
        IF-FAIL:: <- (BECOME RESULT:: <- 0)],
       [(BECOME ((AMOUNT:2 THE) FIXUP) <-
                (DIFFERENCE ((AMOUNT:2 THE) PROTO)
                            (RESULT *RETR)))],
       [*COMP=(COMPUTE AMOUNT:1)
        REQUIRE:: <-
         (ULTIMATE-ARG:: <- (AND (AMOUNT:1 THE)
                                 ((AMOUNT:2 THE) FIXUP)))],
       [(BECOME (AMOUNT:1 THE) <- (RESULT *COMP))]]
```

METHODs are the plans for refinement of algorithms in the programwriter. They correspond to the rules in a production system such as PSI. When a node in the

[10]The basic ideas for the form of these structures come from the form of Owl METHODs ([Sunguroff 1976]), but they have been modified to meet the needs of a programming environment.

program design tree is refined, a METHOD is found that matches it. The METHOD then provides steps to accomplish the purposes of the node. The root concept and pattern for this METHOD is (METHOD (FIXUP AMOUNT)). Any node that needs a METHOD will first have to match the specializer, (FIXUP AMOUNT). Often, the root concept of a METHOD is not as specific as it needs to be to accurately specify what nodes it can handle. Because of this, pattern matching must be a more involved process than just matching the specializer of METHOD. In particular, the pattern matcher also checks that there are appropriate values for the SEMANTIC-CASEs on the node to correspond to those on the METHOD.

The METHODs have variables which are used to specify the places in the refinements for the parts of the node. These variables specify the parts of the node they represent by being in the appropriate SEMANTIC-CASE value in the METHOD. The variables in this METHOD are AMOUNT:1 and AMOUNT:2, the values of the SEMANTIC-CASEs OBJECT and FOR. Thus, the AMOUNT that is the OBJECT of the node will be used as the specializer of the third step and as part of the REQUIRE for that step. The AMOUNT that is the FOR of the node will be used as the specializer of the first step, besides being used in the other two steps. It may be noticed that the variables are sometimes specialized by THE. Specialization by THE or the use of the SEMANTIC-CASE instead of the variable (e.g., if OBJECT: had been used in place of (AMOUNT:1 THE)) is just used as a signal to the pattern matcher. More about that later -- it does not change the way the variable is used once the METHOD has been selected.

The SEMANTIC-CASEs in a METHOD are used to specify the agents that will take part in it. Since METHODs and the other structures may be matched to a variety of nodes, it is important to have a general terminology for the major parts. That is, it must

be an OBJECT in the structure that fills the same role as OBJECTs in similar structures. The SEMANTIC-CASEs provide a set of names suggesting the purpose of the parts. The SEMANTIC-CASEs used in the programwriter include OBJECT, RESULT, FOR, TO, AS, FROM, and IN. The OBJECT is special in that it refers to the specializer of the node. The RESULT is special and refers to the step of the METHOD that returns the result (the primary purpose of the METHOD). The rest of the cases are used to refer to parts involved in METHOD in approximately the way they would be refered to in English. The primary consideration being that METHODs that are related use the same cases for objects serving the same purpose. The values of these cases are variables or concepts containing variables. A variable is a concept (other than a SEMANTIC-CASE or other part of the METHOD syntax) specialized by the METHOD concept. The genus of the variable specifies the kind of concept that can match it.

The part of the METHOD that provides the refinement of the algorithm is the STEPS property. This consists of a set of patterns from which to build the refinement nodes, connected by the primitives of control structure. In this METHOD there are four steps: (RETRIEVE ...), (BECOME ...), (COMPUTE ...), and (BECOME ...). These are connected by THEN, indicated by the commas. There is also a fifth concept, the (BECOME ...) that is a value of the IF-FAIL, that may become a step. It will be a step if analysis determines that it is possible for the (RETRIEVE ...) step to fail. When the METHOD is used to refine a node, these steps are evaluated in terms of the situation existing at that point in the program and become the new nodes of the program design tree.

The steps themselves are of two varieties, calls to other activities and declarations of new facts. The calls all are kinds of ACTIVITY, such as RETRIEVE or ASK, and imply some further selection of METHODs will be needed in refining the resulting

nodes. The declarations have the genus BECOME and indicate that at that point in the METHOD the fact becomes true. The BECOMEs are used in a couple of ways. In the METHOD illustrated there are three BECOMEs, declaring the RESULT of the (RETRIEVE ...) step to be zero, declaring a value for a specialization of the AMOUNT:2, and declaring the RESULT to be the new value of the AMOUNT:1. These BECOMEs may result in a variety of different kinds of realizations depending on the nature of the quantities. It is up to analysis and the data model to decide whether it is appropriate to store the quantity, use it directly in the program, or just use it to answer design questions. Declarations like these are the most common use of BECOME, but it is also used to declare the existence of new properties and new states. For example, BECOME declares the beginning of an I/O statement.

There are also two specializations of METHOD used by the programwriter, PREMETHOD and CODE-METHOD. PREMETHODs are internally generated METHODs used by the argument and control model to incorporate added nodes into the control structure. They are made for specific nodes and do not exist in the initial knowledge base. CODE-METHODs are used by the target language model to turn the final levels of refinement into Lisp code. The only difference is that their steps are made up of LISP concepts and BECOMEs.

```
[(CODE-METHOD (L-RETRIEVE ENTITY))
 OBJECT: <- [ENTITY: (SECOND-PART (DOTTED-PAIR: THE))]
 FROM: <- DOTTED-PAIR:
 STEPS: (LISP CDR DOTTED-PAIR:)]
```

To retrieve an entity that is the second part of a dotted pair, perform the Lisp expression (CDR -dotted-pair-). These LISP concepts, such as (LISP CDR DOTTED-PAIR:) are turned into more primitive forms used as the terminal nodes of the program design tree, and finally into the program by the coder.

There are also a small number of METHODs used to run the top level loop of the programwriter. These are actually executed by the interpreter, rather than being used for refinement. They call the analysis, planning, and coding routines. They are very simple, mainly providing a framework for evaluation.

6.2 Schemas

There are operations on the program design tree that the programwriter needs to do for which the METHODs are insufficient. The METHODs provide a refinement for a particular node. However, sometimes it is necessary to refine more than one node, or refine a data structure that takes part in more than one node. SCHEMAs provide this capability. SCHEMAs are similar to METHODs in form, but are executed instead of being used as patterns for refinements. They are used by the IMPLEMENT module and the CHECK-IDEAS module to do *implementations*, that is, to make modifications to the program design tree. SCHEMAs are used for two main purposes: to refine the data structure, that is, to implement the data in a more concrete representation; or, to change the algorithm in some way by changing or adding operations. The reason a SCHEMA is used to refine a data structure is to insure that every operation using the data will also be refined.

The capabilities provided by a SCHEMA are an extension of those that could be provided through a macro expansion language. A macro takes as arguments all of the pieces of structure it will affect. It makes its additions and changes. Then it is up to the rest of the system to detect and handle any ramifications of the changes. The SCHEMAs are triggered to handle some particular modification, but it is not necessary to provide it with all of the structures it may need. A SCHEMA can specify *locations* in the

program design tree by patterns using the terminology provided by the different models. These patterns can be evaluated, providing the specific nodes needed by the SCHEMA. SCHEMAs, since they are interpreted and not just used as patterns, can also call other modules for information and set up the appropriate goals to handle the possible ramifications of the changes. Many of the aspects of program design are procedural. For example, there is a procedure to turn the computation of a total into a computation that utilizes the previous computation -- a subtotal operation. There are certain steps required to make this transformation, which may involve all of the nodes that either use of change the total. Such a procedure can be specified by a SCHEMA. To do the same thing with macros would require the needed nodes to have been gathered together by the system in some way before the macro could be activated. However, the logical place for the information to reside is in the macro, inaccessible until after the macro has been activated.

One of the main uses of SCHEMAs is in implementing the data structures. The following is the SCHEMA for converting a DATA-BASE into a property on a property list:

```
[(SCHEMA (PL-PROPERTY DATA-BASE))
 OBJECT: <- DATA-BASE:
 RESULT: <- [PL-PROPERTY:
            PL-INDICATOR:: <- DATA-BASE
            PL-NAME:: <- (NAME (DATA-BASE: THE))]
 STEPS: (BECOME RESULT:),
       (BECOME ((DATA DATA-BASE:) THE) <-
             (VALUE PL-PROPERTY:))]
```

The specializer of SCHEMA is somewhat different from the specializer in a METHOD. It is in the form (-to- -from-), indicating the primary transformation made by the SCHEMA. The "from" is the OBJECT of the SCHEMA and the "to" is the RESULT. In this case it implements the DATA-BASE as a PL-PROPERTY whose PL-INDICATOR (the indicator on the property list) is DATA-BASE (which will become a symbol in Lisp), whose PL-NAME

(atom having the property list) is the NAME of the DATA-BASE. It also specifies that the VALUE of the PL-PROPERTY is the DATA of the DATA-BASE. A SCHEMA can also have other SEMANTIC-CASEs besides OBJECT and RESULT, but that only occurs in SCHEMA that are normally called from other SCHEMA. The steps of the SCHEMA are like the steps of a METHOD. The module needing the SCHEMA calls on the interpreter to carry out these steps to accomplish the transformation. This SCHEMA is rather simple, because all of the nodes that will be affected by the change will be detected and handled through the normal operation of the analysis routine. If that were not the case, additional steps could be added to handle those nodes.

The kinds of steps in a SCHEMA are as follows:

1) DO is used to insert a section of code into some location in a program. The specializer of the DO is the code, which may be one step or several steps connected by THENs, ANDs, and IF-THENs. A necessary part of a DO is a LOCATION property to specify where the code is to be added, which may be BEFORE, DURING, or AFTER specialized by a pattern for the desired nodes. The places where the code should be added are all of the places matching the pattern.

2) BECOME is used to specify the things that are changed. BECOME takes an assignment of the form (BECOME X <- Y) and transforms the item bound to X into Y. It can also take the RESULT and will turn the OBJECT into the RESULT. X and Y can be data representations as in the example or nodes in the design tree.

3) IMPLEMENT is used to call other SCHEMA. The specializer of IMPLEMENT is used to search for a matching SCHEMA, which is than interpreted. This is the case where other SEMANTIC-CASEs may be used to pass information.

4) Calls to other routines, in particular Lisp routines for assessing the features important to the SCHEMA. This is done by simply making the call to the desired routine. This feature is not needed by the scenarios.

5) GOAL is used to set up goals that may be needed as a result of the transformations. Doing a transformation on the existing program structure may have ramifications on various other parts of the programs. Thus, it is necessary to be able to set up

GOALs to handle these possibilities. The GOAL statements of a SCHEMA consist of the genus GOAL specialized by the unevaluated statement to be turned into a GOAL. The resulting GOALs are not tackled immediately, but are treated in turn like other GOALs.

With these constructs the SCHEMAs are able to accomplish any transformations needed by the programwriter. The SCHEMAs will need some additional support from the programwriter, in particular from the evaluator and question answerer in resolving questions about the program structure, such as the resolution of location patterns.

6.3 Intents

The INTENTs consist of the information about the nodes they match required to analyze the node. For example, the INTENT for the node in the introductory example is:

```
[(INTENT (ACCEPT (ACT TRANSACTION)))
 OBJECT: <- (ACT TRANSACTION):
 SOURCE: <- OBJECT:
 INPUT: <- OBJECT:
 KNOW: <- ((ETIME OBJECT:) <- ETIME:)]
```

The genus is INTENT and the specializer is a generalization of the nodes it might be used for. The OBJECT and any other SEMANTIC-CASEs can be used to further specify any restrictions on the nodes it can match. The rest of the properties are declarations of facts about the node likely to be needed by other parts of the programwriter. This INTENT accomplishes three things, declaring that the program is a SOURCE for the transaction, declaring that it INPUTs the transaction from the console, and declaring that the time of the transaction is the same as the time of acceptance.

The kinds of declarations that can be made in an INTENT are as follows:

1) SOURCE indicates that the node will be a source for the datum. That is, some node below is a BECOME that declares the datum to officially exist. This information is important to both the data model and the argument model.

2)ARGUMENT indicating what the node expects to be available as an argument. The argument model uses this, but it also indicates what will be used -- information for the data model.

3)RETURN indicates what the node can be expected to make available for other nodes within the program. Notice that this is different from SOURCE.

4)INPUT indicates the node will INPUT something from the console -- used by both the I/O model and the data model.

5)OUTPUT indicates the node will OUTPUT something on the console.

6)Other information includes KNOW and REQUIRE statements giving properties of the data associated with the node and GOAL statements to set up goals that need to be handled before a METHOD can be selected for the node.

It is also possible for an INTENT to have steps, although it is not usually necessary. The steps would call any special analysis procedures that this kind of a node might require -- just a way to call infrequently used parts of analyze. The INTENTs and the METHODs are not necessarily in a one to one relationship. An INTENT provides a means for determining the important properties and requirements of a node, while a METHOD provides one way of accomplishing the purpose of the node. Thus there are times when the KNOW or REQUIRE properties belong on the INTENT because they apply to all such nodes and times when they belong on a METHOD because they are peculiar to it.

The information on the INTENTs is basic to the program design process. Different pieces are used by different models, but it is so consistently needed that it should be readily available on the nodes. It is also information about the purpose of the node and readily available in this way.

6.4 Definitions

DEFINITIONs are used to specify information about the relations between concepts that could not be specified on the generic alone. That is, whenever a variable is needed to specify the relation. The most common uses for DEFINITIONs are to define the calculation of a quantity, such as the definition of the balance of an account in terms of the debits and credits, and to define the relation of SUBSTANTIVE-CHARACTERIZATIONs to other concepts. For example, DATUM which was used in the (METHOD (ASK DATUM)) (see above) is a SUBSTANTIVE-CHARACTERIZATION standing for those items that can be used as independent pieces of data. They include the transactions (such as A-DEPOSIT) and also amounts that are characterized as DATUM, such as the (BALANCE ACCOUNT). The definition of DATUM is as follows:

```
[(DEFINITION DATUM)
 OBJECT: DATUM:
 RESULT: (OR AMOUNT: (ACT TRANSACTION):)]
```

((ACT TRANSACTION) is on the genus path of A-DEPOSIT.) The actual specification of the relationship is the RESULT of the DEFINITION. A DEFINITION for a SUBSTANTIVE-CHARACTERIZATION does not completely specify the entities that have that characteristic. Since any concept has to have its characteristics declared anyway, the DEFINITION provides a pattern for the desired concepts, similar to the pattern for a METHOD and relies on the characteristics to provide the fine specification. In this case all (ACT TRANSACTION)s are DATUM because [(ACT TRANSACTION) DATUM], but not all AMOUNTs are DATUM. The (BALANCE MY-ACCOUNT) is a DATUM because [(BALANCE ACCOUNT) DATUM] and MY-ACCOUNT is a specialization of ACCOUNT.

Other examples of SUBSTANTIVE-CHARACTERIZATIONs are as follows:

TOTAL	a kind of AMOUNT
LISP-ENTITY	LISTs, SYMBOLs, NUMBERs, and AMOUNTs
DATE	NUMBERs that are ETIMEs or DATEs
QUANTITY	NUMBERs that are COUNTs
DB-DATUM	DATUM or (PART DATA-BASE) that is SPECIFIC
PROPERTY	ENTITY that is a part of a DATUM

These are useful for specialized METHODs, IDEAs, and expressing the relationships between implemented data structures.

DEFINITIONs for quantities that can be calculated are very similar, having the RESULT specify a pattern for the arithmetic expression relating the quantity to the arguments. They will be discussed in the next chapter.

Section 7 The Program Design Tree

To design programs it is necessary to maintain in some way the information that is being produced about the problem. The form of such information determines to a large extent the range of capabilities of the design modules and the ease with which they can perform their tasks. In the programwriter are two different kinds of structures being built in the knowledge base during the process of design, the program design tree to represent the state of the program design and the EVENT structure that represents the execution history of the design process itself. The EVENT structure of the design process is handled by the interpreter, representing the progress of the main loop and is the basis of the programwriter's ability to make hypothetical structures.

The program design tree is patterned after the event structure of an executing program. The nodes are also considered to be EVENTs (since they do represent potential future events) and are connected to each other by a SUBEVENT link or an EVENT-CODE link. These links join the nodes into a tree structure for each of the

programs in the request, starting at the node generated from the original call for a program in the request and terminating in the EVENT-CODE nodes representing the Lisp code that will go into the program. They differ from normal events in that there may be alternative paths through them, complicating interpretation. An EVENT is generated by evaluating the call for the event, specializing it by a generated number to insure that it is a unique specific concept, and linking it into the rest of the EVENT structure. The call may be either in the original request, generated by the programwriter, or most commonly a step of the METHOD for the higher level EVENT. An EVENT is connected to the call that formed it through the link CALL, to its INTENT through the link MPLAN, to the METHOD to accomplish it through the link PLAN, and to the superior and inferior EVENT through the link SUBEVENT or EVENT-CODE.

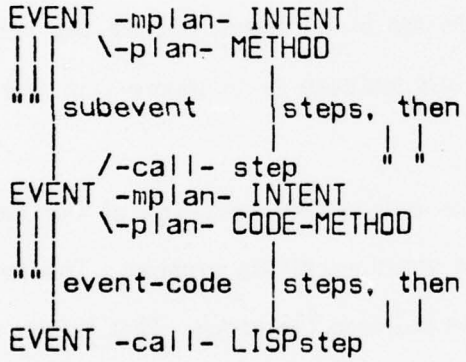

Figure 9. Form of program design tree

In particular, the structure that would have been generated by the end of the introductory example is as follows (the use of colons in this example is not consistent with notation, but the meaning should be evident):

AD-A047 595 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
A PROGRAM WRITER.(U)
NOV 77 W J LONG
N00014-75-C-0661
UNCLASSIFIED
MIT/LCS/TR-187
NL

2 OF 3
ADI
A047595

```
top event:
((ACCEPT A-DEPOSIT*1) 1000001)
   CALL: (ACCEPT A-DEPOSIT*1) ;the concept in the specification
   MPLAN: (INTENT (ACCEPT (ACT TRANSACTION)))
   PLAN: (METHOD (ACCEPT DATUM))
        STEPS: *ASK=(ASK DATUM:),
               *BECOME=(BECOME (DATUM: THE) <- (RESULT *ASK))
SUBEVENT:
   ((ASK A-DEPOSIT*1) 1000002)
      CALL: *ASK
      MPLAN: (INTENT (ASK DATUM))
      PLAN: (METHOD (ASK DATUM))
      ...
   ((ASSERT A-DEPOSIT*1) 1000003)
      CALL: *BECOME
      MPLAN: (INTENT (ASSERT DATUM))
      PLAN: (METHOD (ASSERT DATUM))
      ...
```

Figure 10. Detail of a program design tree

The numbers used as specializers are on all nodes, but for the rest of the presentation they will be ignored. Their only purpose is to insure that the node is unique.

The program design tree contains both a representation of the refinement structure of the program and of the control structure of the program. The refinement structure is indicated by the SUBEVENT links between the nodes. That is, the resulting program will not have a subroutine that calls two other subroutines to ask and to assert. It will have in it operations that are the further refinements of ask and assert. The basis for the control structure is in the METHODs for each of the nodes. How the control primitives interact to produce the final control structure will be discussed later.

7.1 Goals and Ideas

Goals specify what the planning module and to some extent the analysis module need to do. Goals have as genus one of the various classes of goal. The specializer and any properties are as required by the particular kind of goal. For example, (SELECT (METHOD (ACCEPT A-DEPOSIT*1))) is a goal to select a METHOD for the top node of the accept deposit program. Many of the goals require properties of various kinds, which are placed on the reference list. The main difficulty with goals is redundancy. The same goal may be required by several different areas of the program design. Each of these tries to create a goal, but goals are not specialized to make them unique. Thus, each one creates the same goal, just adding any properties peculiar to its circumstances. That way there is one goal for a particular task and it can collect the properties of each area desiring the goal. However in some cases, it is possible for a goal to be needed again after it has been completed. This requires a new concept, so the old completed one is specialized. The resulting goal serves to collect properties until it too is completed. Thus, the algorithm to create a goal becomes: create the concept, if it has the characteristic COMPLETE, specialize the goal and check again. The goals are located on a goal list maintained by the programwriter. The goal list is not inherently ordered, although it provides a default ordering if none other is specified. The primary order constraints are produced by linking goals with BEFORE.

```
[(MAKE (PREMETHOD (STATE (BALANCE ACCOUNT*1))))
 (BEFORE (SELECT (METHOD (STATE (BALANCE ACCOUNT*1)))))]
```

This means that the (MAKE PREMETHOD) goal must be handled at some time prior to the handling of the (SELECT METHOD). If goals are not constrained in this way, they are ordered by types with (MAKE PREMETHOD) coming before (EXPAND DEFINITION) and so forth. Only the order of unconnected goals of the same type is left to the goal list order. When a goal is completed, it is removed from the goal list.

IDEAs are suggestions in the knowledge base for ways of making possible improvements to the algorithms. They exist as properties on the generics of ACTIVITYs and data concepts to which they might apply. Because many IDEAs apply to algorithms only under special circumstances, the use of SUBSTANTIVE-CHARACTERIZATIONs in IDEAs is important in limiting the attempts to use IDEAs to places where they are likely to apply. The IDEA to try keeping a running total (incremental total) of an amount instead of keeping around a subtotal is in the knowledge base as follows:

```
[(DEFINITION (IDEA (SUBTOTAL AMOUNT)))
 OBJECT: (IDEA (SUBTOTAL AMOUNT:))
 RESULT: [(IMPLEMENT (SUBTOTAL (AMOUNT: THE)))
          AS: (INCREMENTAL-TOTAL (AMOUNT: THE))]]
```

The definition is used to provide the variable. If this IDEA structure is found when analyzing (SUBTOTAL (BALANCE ACCOUNT*1)) it is turned into an active IDEA on the idea list, it has the following form:

```
[(IMPLEMENT (SUBTOTAL (BALANCE ACCOUNT*1)))
 AS: (INCREMENTAL-TOTAL (BALANCE ACCOUNT*1))]
```

IDEAs on the idea list are handled by the CHECK-IDEAS module, which is called via a goal. Sometimes an IDEA can be eliminated early, because the SCHEMA to accomplish it does not fit the situation, and sometimes it must be completely implemented to see whether it is an improvement.

Section 8 The Service Modules

The programwriter uses three of the modules of the interpreter to accomplish much of the work of designing. These are EVALUATE, WHETHER, and SUITABLE-REPRESENTATION. These are modules that exist in the Owl interpreter ([Sunguroff 1976]), but have been modified for the purposes of the programwriter. They are called

by many of the other modules to provide evaluated concepts, answer questions, and check for matching patterns, respectively. EVALUATE and WHETHER will be discussed in general here and SUITABLE-REPRESENTATION will be discussed along with METHOD selection.

8.1 Evaluation

Evaluation takes a concept with variables in it and produces a concept by the variables replaced with the concepts they currently stand for. That is the basic purpose -- in practice there are a number of problems with determining the current binding of a variable. It may be bound only implicitly or not at all. The meaning of variables here is also more general than just the variables of structures. It includes the entities that have been changed by a SCHEMA and special concepts called ACTIVEs. Evaluation is used by the programwriter to create program nodes and to instantiate definitions, ideas, goals and so forth. The major difference between the evaluator used by the programwriter and the one used by the Owl interpreter in other systems is the ability to handle FUTURE time. That is, it must handle events that are being anticipated in the program design tree -- it has a *design* problem besides the *execution* problem. In an executing program, past and future are well defined. The results of something that has already taken place is known and available, and the result of a computation is found by doing the computation. In the design environment, many of the evaluations are taking place in the context of the *future*. That is, the results of something that happens earlier in a method is only known to be a result. Or, a computation can at best be only partially evaluated.

The evaluator must operate in this *future* state whenever it is evaluating concepts for the program. For example, a node of the program design tree stands for

the situation that will exist in the program when execution reaches that point. Therefore to produce a node, the evaluation must be with respect to that future situation. On the other hand, some of the evaluation done by the programwriter is for the processing itself. For example, the interpretation of a SCHEMA is a process that takes place during the programming. The evaluation required to determine what the SCHEMA wants done (i.e., the EVENT nodes created in the execution history of the programwriter) are done in the present. In a SCHEMA the DO constructs have locations which must be evaluated. They are locations in the program design tree and therefore treat the structure as an object to be handled. Once a part of a SCHEMA has been evaluated it may indicate that some new node should be added to the tree. The creation of that node will require the evaluation of the concept in the future.

First, it is appropriate to make a distinction. The process normally though of as evaluation in Lisp is divided into two parts in the interpreter (for both Owl and Owl-I). The resolution of variable bindings is handled by EVALUATE. The other job is finding the values of properties, corresponding loosely to application in Lisp, which is handled by FUNCTION-EVALUATE. The use of FUNCTION-EVALUATE is more restricted in the design of programs than in the execution, because information about the source of a value is more important to the design process than the value. For example, the evaluation of AMOUNT: in a METHOD might result in ((BALANCE ACCOUNT) INITIAL). Doing a function evaluation on this concept would produce the value it is specified to have, say zero. If the zero were immediately included in the evaluated concept in place of AMOUNT:, it would be difficult to determine where the initial balance was used.

The most basic problem of EVALUATE is the evaluation of variables, corresponding to symbol evaluation in Lisp. It is the same for either design or execution.

There are three possibilities: 1) The variable is bound. EVALUATE returns the binding. 2) The semantic case in which the variable serves is bound further up the program design tree (see figure 11). EVALUATE returns value of the binding further up the tree. (Thus, DATUM:2 will be bound to A-DEPOSIT*1 in figure 11.) 3) There is no suitable binding. EVALUATE creates a dummy from the variable and the current event on which to keep anything discovered about the concept. Such dummies can later be bound to actual entities if one is found or an actual entity "created" from it as in the case of data representations.

```
  |
node FOR: <- DATUM:1 -binding- A-DEPOSIT*1
     IN: <- (PART DATA-BASE): -binding- DB-DEPOSIT
  |
node FOR: <- DATUM:2
```

Figure 11. Evaluation by event inheritance

The EVALUATOR must also be able to evaluate a semantic case of some other call. The most common situation is the evaluation of the result of some previous call. For example, in the METHOD for (ACCEPT DATUM), the (RESULT (ASK DATUM:)) is needed for the BECOME step following the (ASK DATUM:) step. The desired concept must represent the entity that filled the semantic case in the event for that call (i.e., that was the RESULT in the event for (ASK DATUM:)). To get at that entity the evaluator finds the appropriate event node for the call. Starting with that node, it goes through the same procedure as before to find the correct value. The only catch is that the node of the call may not have a METHOD yet and therefore not have the semantic case available. The programwriter is not guaranteed to complete all of the nodes in order because of the requirements of goals, so it may be that the desired node is awaiting some other action. Because of this the evaluator is capable of causing the

analysis to be suspended awaiting the completion of the needed node. Once the event node is ready for evaluation to take place there is still a special problem of evaluation in a design environment. If evaluation just returned the concept representing the value of the semantic case, the information about the origin of the value would be lost, yet to the routines that will be producing code from this structure that is the most important part. What the evaluator does is return the value specialized by the semantic case of the specific event it will come from. For example, the result of the evaluation of the RESULT above would be (AMOUNT (RESULT (ASK A-DEPOSIT*1))), where (ASK A-DEPOSIT*1) is the previous node. That way the kind of item is known for matching and question answering purposes and the source of the item is known.

Besides the evaluation of variables, there are other kinds of concepts that present special considerations for the evaluator. These concepts include: active concepts, arithmetic expressions, implementation values, substantive characterizations, and values. Active concepts are concepts with the characteristic ACTIVE. This characteristic is used as a signal to the evaluator that the concept requires special processing during evaluation. An example is the STATEMENT concept used by the I/O model to produce the symbols the programs will print to the user (see section III.5). The evaluation of a STATEMENT concept requires specific knowledge about the nodes around it and the specification to produce something to be printed that will have meaning for the user and will observe the format conditions. STATEMENT has the characteristic ACTIVE. The evaluator fills in the variables for an active concept, as it normally would, then it dispatches on the concept to an evaluation routine that claims to be able to handle it. The evaluation routine can then do any additional processing that is necessary.

In an executing environment, arithmetic expressions are ACTIVE and would

just be computed and the result returned. In the design environment the values of the arguments may not be known. Even if they are known the information about where they come from is more important than the value. Therefore, arithmetic expressions are not ACTIVE in the design environment. The job of setting up the necessary calculation for an arithmetic expression is left to the argument handler in analyze.

When an implementation has taken place as the result of a SCHEMA, the evaluator is responsible for substituting the new concepts for the old. For example, when a DATA-BASE is converted to a PL-PROPERTY, the calls that required the implementation must be turned into new calls in terms of the PL-PROPERTY representation. This is accomplished by providing the evaluator with a translation list of concept conversions. This same mechanism is used to handle the SUBSTANTIVE-CHARACTERIZATIONs that will be encountered while dealing with ideas and definitions.

Sometimes the programwriter needs to determine the actual value of a concept, such as when the value must be inserted into the final code, or when a location must be found for an addition or correction to the nodes. Finding values happens in two ways (in both design and execution). Under ordinary circumstances the value is on the reference list of the concept, and can be found by checking the reference list for the most recent value that is the result of a valid event. Many kinds of concepts have special procedures for finding their values. These procedures are found with the dispatch mechanism and are applied to the concept in question.

One more note about evaluation: In the process of evaluation it is possible for the specializer of the concept being evaluated to take on sufficient properties to make the genus be less specific than it could be. Once an animal is known to be warm-blooded and fur covered, it should be considered a mammal instead of just an animal.

The same can happen during evaluation, but the evaluator needs some help from the knowledge base to determine that the genus could be more specific. In particular there must be enough information about the more specific concept to determine whether or not a candidate concept fits. This is handled by the characteristic DETERMINED. If a generic concept has DETERMINED on the reference list, it means that the rest of the properties on its reference list are sufficient to establish whether or not something that is a kind of its genus is a kind of the concept. For example, INSERT is an ACTIVITY that is a kind of ASSERT for something that will become an ELEMENT of a data structure. INSERT has the characteristic DETERMINED, because the only other property, the fact that its OBJECT is an ELEMENT of its IN, is what distinguishes it from other ASSERTs. When the evaluator is completing the evaluation of a concept, it checks the concepts below the one it is completing to see if any are DETERMINED. If so, the properties are checked to see if the result of the evaluation should be given a more specific genus. This is important in keeping the nodes as specific as possible to ease recognition of the situation.

8.2 Whether

Besides the need to evaluate concepts, the programwriter must be able to handle the predicates in the specifications, METHODs, and SCHEMA. WHETHER is the predicate evaluator for the programwriter. It has the responsibility for determining the truth of the predicates in IF-THENs and is called by other modules to answer their questions. WHETHER operates through a dispatch mechanism like that for FUNCTION-EVALUATE. The procedure found through the dispatch on the predicate is applied and may in turn require other calls to WHETHER. The major difference between WHETHER in

a design environment and WHETHER in an execution environment is the logic needed. In design all of the possible outcomes of the predicate need to be considered. This is handled by having four possible results for WHETHER: TRUE, FALSE, EITHER, and DON'T-KNOW. TRUE means that any time in the future the result will be true. FALSE means the result is always false. EITHER means the predicate can be either true or false at times during the execution history. DON'T-KNOW is WHETHER's way of suspending. It is accompanied by a goal and the WHETHER will be suspended until the goal has been completed. It is not always easy to determine for sure whether something is always true or false. Since EITHER is the most general answer and is always safe, the whether procedures err in favor of it. If EITHER is not the best default answer, the predicate can indicate the appropriate result as an ERR property.

There are several different kinds of requests that are handled by WHETHER procedures. These include the arithmetic predicates, questions of how items are characterized, existence, emptiness, needs, legitimate values and so forth. The more important of these will be discussed with the modules that use them. Many of the uses of WHETHER involve checking for characteristics that would be left on the concepts by other procedures. However, some involve more complex reasoning and will be discussed in chapter five after there are some examples to show the problems.

Chapter III
Models for the Programwriter

The programwriter makes use of five different models of the programs during the design process, besides procedurally representing a model of the design process itself. The programwriter is engaged in a design effort. To carry out the task of designing programs, it must be operating in accordance with some model of how designing happens. Several modules of the programwriter carry out functions for that model exclusively. To organize the design process the five models of programs are represented by modules to aid in the design process while maintaining their own view of the program. The models are not separate monolithic entities but are integrated into the programwriter as small chunks in three forms: some are structures in the knowledge base; some are modules to carry out the planning activity; and some are specialized versions of the general functions used by the programwriter. The models are split up in this way to conform with the breakdown of tasks provided by the model of design. The interfaces between the models are in the form of terminology shared by different models and modules of one model that are called by other models. A representative sample of these interfaces is shown in figure 12. (The boxes represent modules. The lines represented terminology directed from source to use.)

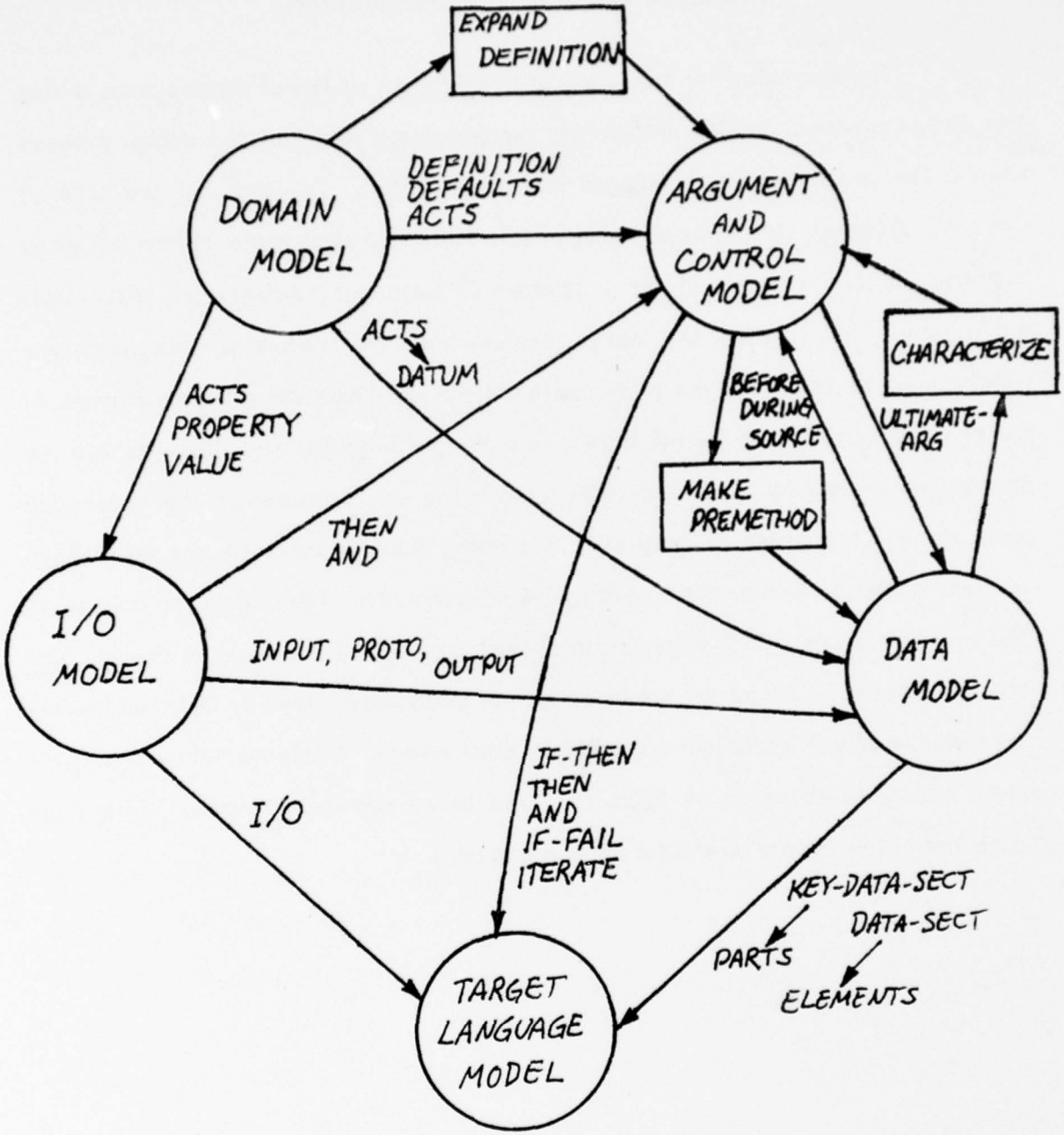

Figure 12. Model Interfaces

Each of these models has a set of tasks to carry out and represents a view of the programs suitable for carrying out those tasks. This chapter will present these models, the tasks they are to carry out, the particular terminology and problem space they work in, and how they operate in relation to the programwriter and the rest of the models. First however, the model of design and the modules it uses will be discussed.

Section 1 The Model of Design

The model of design used by the programwriter is a hybrid of successive refinement and modification for improvement. The basic driving principle around which the programwriter is built is refinement. All of the structure built up in the last chapter provides an environment in which refinement can take place. But, the open accessible structure of the program design tree is also available for other purposes, such as analyzing data dependencies or determining relationships between nodes. It provides a continuum from the specification to the program, which in turn provides a space in which the second design technique can operate. This second technique is recognition and modification. It utilizes recognition of situations during analysis to find IDEAs, which are suggestions for modifications that might improve the program. These modifications can then be tried and evaluated.

1.1 Refinement

The programwriter operates primarily by successively refining the specification it is given. Refinement means taking a requirement of the current level of abstract description and designing a less abstract way of satisfying the requirement

utilizing the knowledge provided by the rest of the description. The refinement can be compared to macro expansion, but the way the refinement is chosen is different from normal macro expansion. It is chosen based on the primary distinguishing feature of an operation rather than a pattern for the arguments. The primary distinguishing feature depends on the kind of operation. Consider the refinement of (ASK A-DEPOSIT*1). A-DEPOSIT*1 is not an argument of ASK in the normal sense. Rather, it is main way in which this ASK is different from other ASKs. If the different calls to do ASKing were categorized, A-DEPOSIT represents the attribute of the ASK that would be the basis of the categorization. In this case it will be neither an argument of the code or the result (in a strict sense). The section of code resulting from the call will have no arguments, because it will get what it needs through reading and writing on the console. If the section of code were separated from the rest, the items it "returns" are all of the parts of the A-DEPOSIT*1 needed by the rest of the set of the programs rather than a single structure that could be called the A-DEPOSIT. This "argument", the OBJECT, along with the other semantic cases is used by the programwriter to guide the design process.

Besides refining the task by selecting METHODs, the programwriter also needs to refine the data. This takes place on two levels. First, the structure of the data must be tailored to the requirements of the programs by selecting what the contents will be and the requirements for the identifier. Secondly, the data requirements must be brought down to the machine level by producing a data base representation of the data, and then selecting suitable Lisp implementations of the data base parts. The actual control structure of the programs is decided in the coder after all of the refinement has taken place. Then the appropriate control structure can be chosen on the basis of the semantic structure designed through analysis and planning.

1.2 Control of the Design Process

The refinement process provides the basic control for the programwriter, but the way that control happens is not exactly straight-forward. All of the pieces were in the previous chapter, but it is important to see how they fit together. The first level of control in the programwriter is the analysis and planning loop. Each time through the loop a goal is handled by the planning routine and the results of processing the goal are analyzed. This is the basic cycle of the programwriter with all of the parts fitting into either the analysis phase or the planning phase. A more elusive control structure exists in the way planning operates. There is no sure way of knowing in advance the proper order in which to do the things that must be done to write a program. The steps are often interdependent and it is not until some attempt is made that the dependencies become apparent. The planning routine runs by selecting a goal from the current list, and handling it as best it can. If the handling of a goal leads to some difficulty, the goal can be suspended with a constraint placed on it requiring that it be tried again after the difficulty has been resolved. This kind of situation might take place when a result of some procedure is needed for the current call, but that procedure has not been analyzed yet. The current goal would be constrained to happen after the analysis of the other procedure. This is one way that the situation in the specifications forces changes in the control of design.

It would appear that the programwriter could get into great difficulty with simultaneous goals waiting on each other, however many of the situations that cause simultaneous goals in other systems can be handled "simultaneously" (as a single goal) because there is an appropriate view of the program. If a specialized data structure were to be implemented which required special procedures for entering new elements,

special initialization, and special handling before using, the implementation could all be handled by a single SCHEMA, because the data model provides the proper view and terminology to name all of the locations that will have to change. In an extended version of the programwriter there is the possibility of goal conflicts of the following kind: Goal A could be handled more decisively if goal B were handled first, and conversely goal B could be handled better after goal A. The answer to such a dilemma is to go ahead with one of the goals.

The way that goals are selected by the planning routine is by first following the chains of prerequisites indicated by BEFORE. These constraints are imposed by the INTENTs put on the nodes by analyze as well as by trying the wrong thing first. Once goals have been found that have no prerequisites, they are taken in order of goal class. The order is (MAKE PREMETHOD) then (EXPAND DEFINITION), followed by (SELECT METHOD), CHARACTERIZE, (CHECK IDEAS), and finally IMPLEMENT. There are also goals for the analyzer, but these are ignored by the planner. Beyond this filtering, they are taken in order, which is approximately the order imposed on the nodes by the steps of the METHODs they have. Handling the goals in that order cuts down on the number of goals that discover prerequisites.

Beyond the control provided by the goals, it is reasonable to talk about the set phase, the data base phase, and the Lisp phase of the design. These are the basic steps of refinement according to the data model. The reason the refinement of data structures provides this kind of control is the place of the IMPLEMENT goal at the end of the goal selection list. IMPLEMENT takes care of the actual refinement of data structures and only takes place after the other goals have been completed. This is necessary because implementation requires complete knowledge about the structures

involved to make appropriate decisions. The boundaries of these phases also provide a consistent environment in which to test IDEAs and be able to compare them at an appropriate stage of design.

1.3 Method Selection

The refinements for the nodes in the program tree are provided by selecting a METHOD for the node. The first step in selecting a METHOD is verifying that the node does not already have a METHOD -- easily done. Assuming the METHOD is needed, the procedure for finding one depends on the concept tree. METHOD specialized by the evaluated call is used as a starting place to look for the METHOD. Searching up the tree, each prospective METHOD is submitted to a thorough pattern match against the node including any semantic cases that might be required by the METHOD and any required relationships between the variables to see if it really fits the node. The required relations in a METHOD are any that exist between the variable representations, not including relations that exist between variables specialized by THE or between semantic cases used in place of variables. Thus, if the OBJECT of a METHOD were AMOUNT:1 and the FOR were AMOUNT:2, the following would be required of a node to match the METHOD:

(> AMOUNT:1 AMOUNT:2) required
(> (AMOUNT:1 THE)(AMOUNT:2 THE)) not required
(> OBJECT: FOR:) not required

The order of the search for METHODs through the knowledge base is to proceed up the genus structure, starting by testing the most specific pattern first.

Because the METHOD search relies on the concept tree the organization of the METHODs is important. The specializer of the activity drives this search so it should

be the primary feature used to discriminate between different calls to this kind of activity. Usually, this is in some sense the purpose of such a call, such as what is being retrieved or the meaning of an assertion. The more discriminatory it is the smaller the number of METHODs that have to be checked. There does need to be some compromise, because METHODs of the same type may be different for different reasons while there must still be consistency among the semantic cases. For example, methods of retrieval differ in some cases because of what is being retrieved while in other cases they differ because of what the entity is being retrieved from. For consistency all of the retrieval METHODs have as OBJECT the entity being retrieved. When a METHOD concept is found by searching up the genus path, it may only be a link to the actual METHOD. This occurs for two reasons. If the METHOD uses SUBSTANTIVE-CHARACTERIZATIONs such as DATUM in the METHOD concept, an alternative METHOD concept must be included in the knowledge base for each alternative primary concept of the SUBSTANTIVE-CHARACTERIZATION. That is, there will be a concept (*METHOD (ASK AMOUNT)*) that points to the actual method at (METHOD (ASK DATUM)). Secondly, it may be that the specializer does not discriminate between two METHODs. Because each METHOD must be unique, the actual methods are given meaningless specializations to distinguish them and are pointed to by the METHOD concept that would have been used.

If a METHOD is in the path, it must be fairly close to what is desired. To make sure, it is necessary to do the pattern matching. The specializer of the node must match the OBJECT case variable, which is the same as the specializer in the METHOD concept but more specific. Each of the semantic cases required by the METHOD either has a corresponding case in the call or the evaluator can find one. The concepts on the node must be in the class specified by the generalizer of the method variable and satisfy any relationships specified by the METHOD.

Some of what would ordinarily be called local optimization is accomplished by METHOD selection. For example, there is a METHOD for the addition of zero to a number that just returns the number. This approach is limited to simplifications involving a single node[1], but it means that no other part of the programwriter has to worry about them. Simplification can take place in this way, because METHOD selection is based on the contents of the *evaluated* node rather than the call. That is, the things exist in the particular program situation determine the method, rather than what might have been anticipated by the author of the method that contains the call.

1.4 Recognition and Modification

It may be argued that the best METHOD for a node can not always be chosen on the basis of criteria about that node listed in the METHOD -- a static structure. This is true. One alternative is to be less discriminating, produce all possible structures, and chose among them. This is basically the approach chosen by the PSI project [Green 1976], plus a sophisticated pruning strategy to limit the number of structures actually produced. The philosophy in the programwriter is somewhat different. There is a notion of a *basic program* implied by the specification. That is, utilizing the refinements that seem most appropriate and the data structures that the fit each situation a basic program will result. This can then be considered a basis from which to make improvements. Those improvements are made on the basis of situations recognized in the basic program.

[1] This does not include all local optimizations. Optimizations involving adjacent instructions, such as peephole optimization, may cross node boundaries at several levels. The place to handle such things is in a model that has the appropriate view -- for peephole optimization, the target language model.

The recognition process is carried out by analysis. Whenever a node is to be analyzed, a search is done for IDEAs. Starting from the node, the search goes up the genus path looking on the reference lists for IDEAs. Also, starting from each of the major parts of the node, e.g., the parts that are semantic cases, a search is conducted up the genus path. Some of the IDEAs that are found may not apply. Just as in the case of the METHODs, IDEAs may make use of SUBSTANTIVE-CHARACTERIZATIONs. The IDEA concept that is found may be a pointer to the real concept with the SUBSTANTIVE-CHARACTERIZATION. If that is the case, the corresponding concept in the node must be checked to see that it has that characteristic before the IDEA is returned from the search.

These IDEAs go on the idea list to be handled when the (CHECK IDEAS) goal is handled by the planner. They represent possibilities for improving the code. When the IDEA checker processes an IDEA it may find that it does not apply, or that its suggestion is not an improvement to the program, or that the result provides a better structure. The basic strategy for checking out an IDEA is to first set up a new event marked HYPOTHETICAL, which through the normal binding and value process will be included in any results made as a result of trying out the IDEA[2]. That way, the results can easily be included or not or compared to other HYPOTHETICAL results.

The actual IDEA is a goal to be carried out, usually to IMPLEMENT some change. For example, the IDEA to change a subtotal calculation into an incremental total calculation (section II.7.1) calls on a SCHEMA that adds computations at the sources for the arguments to the calculation and turns the old calculation into a retrieval. The goals

---

[2]This is simple *context* mechanism similar to that provided by CONNIVER ([McDermott and Sussman 1974]), but restricted to this purpose and the handling of simple cases of deduction of the form "If x were true, would...".

for IDEAs are specialized by the IDEA to indicate what it is for. This effectively causes the goal mechanism to recurse. IMPLEMENT goals normally wait until last, but goals for the current IDEA are processed before all others. Once the idea checker has set up the goal, it suspends itself awaiting the outcome of the goal. The process of IMPLEMENTing is simply the running of a SCHEMA that matches the goal. This is the first place that the IDEA might fail. The process of finding the SCHEMA is the same as the process of selecting a METHOD. A search is conducted and a thorough pattern match compares each candidate to the node to be changed. If no SCHEMA matches, the IDEA is discarded. If a SCHEMA is found, it is carried out. That is, the interpreter runs it as a subroutine to modify the program design tree. It may call other SCHEMA and may or may not be successful, depending on the situation. (This process will again be discussed after the scenarios when example situations can be explored.)

After the goals required by the IDEA have completed, the result must be compared to the basic program or the current best modification of it. That does not mean that the basic program has to be completed for the comparison to take place. The comparison is made on the basis of a categorization of the importance of the storage and computation requirements. Increasing (during the execution history) storage demands are considered more important than increasing computation demands, which are more important than sawtooth storage demands, and so forth. The determination is made by comparing the demands in the most important category where the programs differ[3].

Thus, the mechanisms are provided to improve the basic program designed by the programwriter. This might be called yet another way in which the specification is

[3] I do not pretend that method of comparison is the best possible. It is sufficient for the purposes of the thesis. Other people are exploring the ways such comparisons should be made ([Barstow and Kant 1976]).

refined during the design. However to distinguish it from the refining done by METHOD selection and data representation selection, it is called recognition and modification.

Section 2 The Domain Model

The domain model is the source of information about the application area. There are actually three different "domains" that the programwriter needs to know about. One is the domain of programming, which is what the whole thesis is about. All of the models are a part of it. The second domain is the domain of the programs the programwriter will produce. That was the subject of the program environment section, the specification language section, and it permeates much of the rest. Finally, there is the application area for which the programs are being produced. The domain model takes the view that the program is operating in the world of the application area; the parts of the program represent real objects and relations in the application area; and therefore the program must obey the same rules, suitably abstracted, as the real objects and relations. Thus, it is the domain model that is responsible for preventing such problems as selling tickets to events that have already taken place.

This model is somewhat different from the other models in that it does not have a particular view of the internal structure of the program. Instead it provides the information about the application area concepts by providing facts in the knowledge base to answer questions, by providing abstractions of the concepts suitable for programming, by providing definitions for the relationships between concepts, and by providing the terminology and set characteristic facts to compare the efficiency of programs. This knowledge is mostly declarative, but there are also a few evaluate routines (analogous to CONNIVER IF-NEEDED methods [McDermott and Sussman 1974]) and a module for

expanding definitions. The most basic declarative way that information is provided by the domain model is through the categorization of concepts in the knowledge base taxonomy. Besides this, information is provided as characteristics, relations, and definitions on the generic concepts.

The general application domain of the programwriter is a simple problem space of programs to handle the relationships involved in representing simple kinds of transactions in the real world. For example, in the savings account problem the possible transactions are deposits into the account, withdrawals from the account, and requests for the balance from the accounting system. The deposits and withdrawals are events taking place in the real world and the balance is an entity in the real world, all of which must be represented in the programs.

The programs are viewed as abstractions of these real world events and are called on to represent the aspects of the events that are important to the programming process. To find out what these aspects are and how they might be represented, it is necessary to look at the nature of real world events and their relationship to programming entities. For a program to handle deposits to a savings account, consider what a deposit is. Depositing is an activity involving a person, a bank, an account, and some money. By saying that, the activity is already being abstracted from reality, because an arbitrary amount of detail could be included, such as a deposit slip, a teller and so forth. The deposit takes place by the person, who is in possession of the money and owner of the account, going into the bank building (or some branch of the bank) and transfering the money to the bank to go into the account. The program specification calls for a program to represent this transaction, implying that the program will be called each time such a transaction takes place and will provide for any other programs the

information about the transaction needed for their purposes. To do this it is necessary to abstract from the depositing activity a form that captures the parts important for a program.

This abstraction process takes place in three phases: representing the activity in the real world as an activity in the knowledge base, abstracting from the activity structure a general form for a datum representing the process, and finally tailoring the general datum to the specific structure needed for the set of programs (see figure 13). The first two steps of this process are not actually done by the programwriter, but it is assumed that the results are already in the knowledge base. The third step takes place during the designing of the program by determining what parts of the generic datum are actually needed in the given situation.

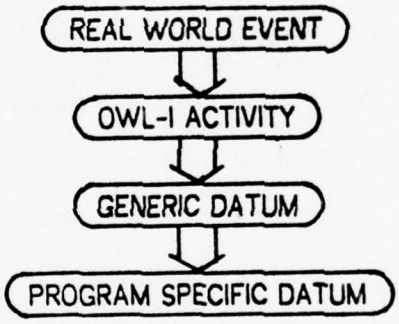

Figure 13. Refinement of an event into a datum

The first step is to abstract from the real world activity a description of the activity that includes the parts generally important in programming. People do this all of the time. It is the thinking of depositing in terms of a person, a bank, an account, and some money, when so much else could be included. The Owl-I method for *depositing*, which is the *deposit* abstraction in the knowledge base is as follows:

```
[(METHOD (DEPOSIT MONEY))
 OBJECT: <- MONEY:
 (DESTINATION MONEY:) <- ACCOUNT:
 (SOURCE MONEY:) <- PERSON:
 [(PERSON: THE) DEPOSITOR]
 (LOCATION ACCOUNT:) <- BANK:
 LOCATION: [BANK-BRANCH: (PART BANK:)]
 (OWNER ACCOUNT:) <- PERSON:
 STEPS: (BECOME (LOCATION MONEY:) <- (DESTINATION MONEY:))]
```

The programwriter must deal with different kinds of activities, making it reasonable to provide some additional specialization of the concept ACTIVITY. The activities, such as DEPOSIT, that are representations of real world activities are kinds of TRANSACTION. With this method, the programwriter is provided with the relationships between the money, the depositor, the bank, and the account. It also specifies the kinds of entities that can fill these positions. In the deposit a person, called a depositor, transfers money in his possession to an account owned by him and located in a bank. The transaction takes place in some branch of the bank. This is the most detailed information about depositing in the knowledge base.

The programs to be written will operate in terms of *data* -- bundles of information representing the occurrence of such an activity. From the METHOD form must come the general form for such a datum. That is, the verb form, the activity (e.g., to deposit), must be turned into the noun form, the act (e.g., a deposit). In English, the noun form of a verb may represent either the event or the result of the event. Here the datum will represent the event. However, if there is reason to refer to the result of the event, that is called the VALUE of the datum. To make such a datum the variable parts of the METHOD which are generally useful in a programming environment are made into properties of the noun form. The noun form is represented by the concept ACT specialized by the activity. All ACTs have one property that does not come from the METHOD -- the time the activity took place, represented by ETIME (for time of

execution). Also, some subset of the properties must be designated the IDENTIFIER. The identifier properties are the properties needed to tell one ACT of the same type from another. More than that, they are the properties used to index the ACTs. Thus, they may include more than just the minimum subset necessary to distinguish between ACTs. Besides the basic properties an ACT may also have information about the frequency of occurrence or quantity of the ACTs or some subsets of the ACTs. This will be useful information when it is necessary to make comparisons between programs.

The ACT for the deposit activity has the following information:

```
[A-DEPOSIT = (ACT (DEPOSIT MONEY))
          IDENTIFIER: <- (AND [ETIME: PROPERTY]
                              [ACCOUNT: PROPERTY])
          [(AMOUNT MONEY): PROPERTY]
          [BANK-BRANCH: PROPERTY]]
```

Thus, the ACCOUNT, the (AMOUNT MONEY) (given the label AMOUNT-M hereafter), and the BANK-BRANCH are considered to be the parts of the ACT important to a programming situation. The characteristic PROPERTY also declares that the user will understand when reference is made to these properties. That is, a program can ask for the "account of the deposit" or "amount of money of the deposit" and be understood. PROPERTY is one of the terms common to both the domain model and the I/O model. There is also information about the frequency of the ACTs, but it is on a specialization of the concept:

```
[(RATE [A-DEPOSIT#1 ACCOUNT:: [ACCOUNT#1 SPECIFIC]])
 (CONSTANT APPROX)]
```

Therefore, for any specific account the rate at which acts of depositing occur is approximately constant over the execution history of the set of programs (and assumed to be significantly greater than zero). The rate could also be specified more precisely as ((PER 1-MONTH)(5 APPROX)). Or, the history pattern of deposits could be specified less precisely as:

[(COUNT (SET A-DEPOSIT#1)) INCREASING]

where INCREASING means increasing over time. All of these measures are very crude, but they only need to be sufficiently precise for the program comparison routine to properly categorize the storage and computation requirements.

The ACT of the deposit has been refered to as a *datum* several times. The programwriter uses the concept DATUM to refer to a complete bundle of information relating to the application. That is, any ACT of a TRANSACTION is a DATUM (hence, [(ACT TRANSACTION) DATUM]) or any amount known by name in the application area is a DATUM (e.g., [BALANCE DATUM]). DATUM provides a handle for the data model to identify the information that may be passed between programs and also assists the I/O model in communicating with the user. The ACT concept provides a starting place for the design of a suitable datum for the programs, an operation which will take place during the design of the programs. It may be discovered that some of the properties are not needed, or even that some additional property is needed.

The ACTs are basic to this programming domain, but there are also other concepts in the specification refering to parts of the application domain. The ACCOUNT and ETIME also have information of their own. The ACCOUNT is a number, with the characteristic NUMBER, making possible a suitable type check when it is input. There is also something known about the behavior of ACCOUNTs over time. The fact:

[(COUNT (SET ACCOUNT)) (CONSTANT APPROX)]

declares that the total number of accounts can be expected to vary only an insignificant amount over the execution history of the programs. That means in particular that after a while most of the deposits and withdrawals received will be for existing accounts. On the ETIME concept there is a NAME property

[(NAME ETIME) <- TIME_OF_OCCURRENCE]

which tells the I/O model how the concept should be refered to for the user.

Besides the PROPERTYs of ACTs there are also concepts such as BALANCE, which specify a relationship between the ACTs. The relation of such a concept to the other concepts in a specification is declared by a DEFINITION. The relationship is specified in as general terms as possible to have wide applicability. For example, the definition of the balance might be as follows:

```
[(DEFINITION (BALANCE ACCOUNT))
 OBJECT: <- (BALANCE ACCOUNT:)
 RESULT: <- (DIFFERENCE (SUM (SET (CREDIT (ACCOUNT: THE))))
                        (SUM (SET (DEBIT (ACCOUNT: THE)))))]
```

That is, the balance of an account is the difference between the sum of all of the credits and the sum of all of the debits. When such a definition is needed for a program, the argument model initiates a search. When it is found, a goal is set up to EXPAND the definition. That is done by the domain model through the evaluator. If this definition were to be expanded for (BALANCE ACCOUNT*1), it would be given to the evaluator with ACCOUNT: bound to ACCOUNT*1. This definition can be evaluated, but CREDITs and DEBITs are SUBSTANTIVE-CHARACTERIZATIONs rather than actual entities in a specification. Money amounts can have these characterizations. Credits include such things as deposits, interest, and the initial balance. Debits include withdrawals, charges, and loan interest. To determine what the credits are in this situation, the appropriate evaluation routine conducts a search of the specifications for entities that match the definition for CREDIT. Since the definition may include entities that are not actually CREDITs, each match is checked to see if it has the characteristic CREDIT. The search in the example to be used in the first scenario will find the amount of money in the A-DEPOSITs and the initial BALANCE. The analogous evaluation of of the DEBITs will find the amount of money in the A-WITHDRAWALs. Thus, the evaluation of the DEFINITION would produce:

```
(DIFFERENCE (SUM (AND ((BALANCE ACCOUNT*1) INITIAL)
                      (SET (AMOUNT-M [A-DEPOSIT#1
                                      ACCOUNT: ACCOUNT*1])))
                (SUM (SET (AMOUNT-M [A-WITHDRAWAL#1
                                     ACCOUNT: ACCOUNT*1])))),
```

where ACCOUNT*1 is the ACCOUNT in the specification for which the definition is evaluated. This is the desired definition that EXPAND-DEFINITION would provide, completing the (EXPAND (DEFINITION ...)) goal. The INTENT and METHOD for (SUM AND) would turn the first argument of DIFFERENCE into

```
(PLUS ((BALANCE ACCOUNT*1) INITIAL)
      (SUM (SET (AMOUNT A-DEPOSIT)))).
```

The resulting evaluated definition is the one that is actually included in the scenario, rather than going through the expansion again.

The domain model also provides initial values for concepts in the application area that might have them. For example, the initial balance of an account is known to be zero. Actually, most of the initial values used in designing programs come from the programming knowledge provided by the other models.

One other responsibility of the domain model is to provide the information necessary to compare the efficiency of programs. The module to do the comparison (a part of the argument model) can tell that a program will take time proportional to the size of some set and the data model can provide information on the relationship of the sizes to the number of ACTs of the programs but it is ultimately up to the domain model to tell the characteristic sizes and time behaviors of the real world ACTs and entities. The metric used for sizes is very simple. Sizes come in three classes: CONSTANT, SAWTOOTH, and INCREASING. These can be given more definite numeric values, but for the cases of interest these are sufficient. These are actually size patterns over time, providing the information that the comparison module needs to make coarse comparisons.

CONSTANT implies the size does not vary over time ((CONSTANT APPROX) is equivalent for comparisons, saying that the size does not vary significantly). SAWTOOTH implies that the size increases to a certain point, then drops only to increase again. The maximum size is bounded. INCREASING means that the size of the set is increasing over time, and is not bounded. This can be deduced from RATE information given about the elements of a set. In general it does not take much information about the sizes of the specification entities to make reasonable comparisons possible. The size of the ACCOUNT set and the rate of the A-DEPOSITs and A-WITHDRAWALs is sufficient to determine the characteristics of the computations and the storage needed in the scenario once the algorithms have been determined.

When the programwriter operates in a different application area, it needs a different set of domain model information. That means the following information about the new area must be provided:

1) All of the entities in the application area must be given concept representations, fitting them into appropriate places in the knowledge tree.

2) The transactions must be abstracted and represented so they can be used as data.

3) Any computable quantities must be given definitions, making it possible to design appropriate computations for the code.

4) The dynamic characteristics of entities in the domain must be specified, that is, the rates at which sets change and any important interdependencies.

5) The definitions of any terms that usefully classify the entities in the application area must be provided.

6) Any default values of entities need to be provided.

2.1 Interfaces with Other Models

As a summary, it is useful to review what the domain model provides for the other models, i.e., what the interface is between this model and the others. The DEFINITIONs provide one of these interfaces. For the domain model, a DEFINITION is a representation of the relationship between concepts, but it is also an interface used by the argument model. The argument model knows that the DEFINITION can provide a template for producing a needed quantity, and uses it as such. Thus, the interface is provided by the concept DEFINITION which has a meaning for both of the models. The concepts representing the ACTs in the domain are abstractions of the real world events to the domain model. To the data model, they are representations for DATUMs, which must be refined, have their storage requirements handled, and be characterized. To the argument model, they are just quantities that have parts. To the I/O model they are things which will have to be input and output to the user. In particular, the parts marked as PROPERTYs are important enough to be understood by just their part name. Thus, at the level of ACT or DATUM or PROPERTY, the terminology provides an interface with the other models. Similarly, the default values interfaces with the argument model and the data model and the sizes provide an interface with the data model.

Another kind of interface between models is in the form of services provided by one model for others. Most of the information provided by the domain model is declarative, but it does provide the EXPAND-DEFINITION module, used by the argument model. This module evaluates the general definition in light of the specifics in the specification.

Section 3 The Argument Passing and Control Model

The argument model views a program as a set of primitive operations connected by the passing of arguments and the basic primitives of control. The relations in programs that are important to this model include the order restrictions on operations, the origin of arguments to operations, conditions for executing operations, and the control of iteration. If there were recursion in the domain of the programwriter, that would also be important to the model.

The model is interested in a static view of a single program and the possible paths through the program. That is, it views the program as a graph of nodes connected by directed arcs. The model is interested in four different kinds of arcs: indicating the result of one node is the argument of the other, indicating that one node must occur before the other because of side effects, indicating a flow of control, and connecting an iteration with the operations occuring inside the iteration. There are also five different kinds of operations of importance to the model: operations which only need to be done once, other normal operations not affecting the flow of control, conditionals, changes of state, and iterations. The purpose of the model is to build, maintain, and answer questions about this view of the program.

The restrictions on the possible program graphs are as follows:

1) The control paths are partially ordered (sufficiently ordered to be deterministic, but not more ordered than is necessary).

2) There are no loops in the graph. This restriction is not unreasonable, because if such loops occurred they could be replaced by iteration constructs.

3) An iteration arc connects the iteration to the body. The body parts are not connected to anything that is not a body part, except for possible exit arcs that return to the iteration node.

4) The additional ordering constraints placed on the structure are consistent with the control paths.

5) An argument arc must connect two nodes consistent with the control paths. That is, if there is a node that takes an argument, that argument must come from some other node. (The programs in this domain are not called with arguments, any needed information is requested as input from the user.)

The model carries out many functions in connection with the requirements for the program structure including incorporating new nodes into the structure, searching the control path, finding or generating nodes to produce required argument, and trimming the control structure to only that needed for the program.

3.1 Control in the Program

First, consider the control structure of a program in the design tree. The control structure is specified by six concepts: THEN, AND, IF-THEN, IF-FAIL, ITERATE, and EXIT. They are specified in the METHODs of the nodes and provide a complete picture of the control structure. The basic sequencing primitives are THEN and AND. THEN means that one operation follows the other. In the METHODs [(THEN A) <- B], meaning do A and then B is abbreviated as [A, B]. (AND A B) means that both A and B are to be done, but the order does not matter. A set of nodes connected with the AND and THEN concepts (without loops) constitutes a partially ordered control path, specifying to the degree required the order of the operations. The basic branching primitives are IF-THEN and IF-FAIL. The IF-THEN construct [(IF-THEN P S1) ELSE: S2] consists of a conditional P, a sequence of steps S1 to follow if it is true, and an optional ELSE clause with steps S2 to follow if it is false. The conditional can be in one of three states: TRUE, FALSE, or EITHER with a characteristic on the reference list indicating the

state. If it is TRUE, the S1 steps are the only ones that apply to the program. The conditional has been declared true for all cases and therefore is transparent. If it is FALSE, only the S2 steps apply. If it is EITHER, there is a node attached to the IF-THEN to (TEST P) that performs the test of the predicate and each set of steps represents an alternate control path. The construct [(IF-FAIL -step-) S3] provides an alternate path S3 if the node fails. The concept also takes a characteristic of TRUE, FALSE, or EITHER indicating whether the control path follows the node, S3, or both. There must also be a (TEST (IF-FAIL -node-)) to determine which branch to take. Both the IF-THEN and IF-FAIL introduce branching into the program if they are EITHER. It is possible to explicitly rejoin the control path by giving to THEN concepts the same value, but the control path normally rejoins through the refinement structure. That is, the section of nodes in which the branching takes place is the method expansion of some node higher in the design tree which may be followed by other nodes. The ITERATE concept takes a SET over which to iterate and performs the BODY of the ITERATE with each successive element of the SET taking the place of (ELEMENT (SET ...)) in the BODY, unless the BODY explicitly exits the ITERATE by using (EXIT (ITERATE ...)). The BODY is a control path separate from the rest of the control paths in the program. The sets over which the iteration takes place are considered to be unordered (the complications of ordering would provide a nice extension).

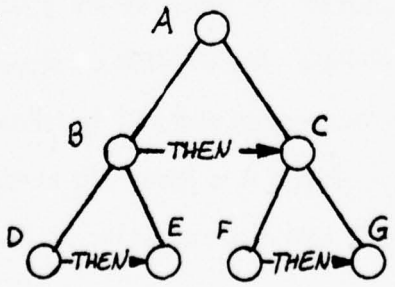

Figure 14. Control structure in the design tree

As mentioned, all of this control structure exists in the design tree among the refinement structure. Thus, there are THENs at different levels of refinement as shown in figure 14. This could be interpreted to mean that the terminal nodes are strictly ordered, D, E, F, then G. However, that is not what really happens. The step B happening before C means that some essential part of B must happen before some part of C. Sometimes this is because an argument for some part of C is supplied by some part of B. This will be analyzed and handled through the argument mechanism (see below). Other times a change of state occurs in B that changes what will happen in C. The changes of state may be changes to the data structures, indicated by BECOME concepts, or the execution of I/O. In any case parts of B and C not involved need not be so tightly constrained. The final decisions on order are made by the coding module after the refinement process in complete and all of the information is available. It is then possible to tell what constraints are actually required. At that time the situations that must maintain their order within the structure are the argument relations, the modifications of a piece of structure relative to retrievals from the same piece of structure, and all of the I/O operations. To assist, the I/O model also attaches BEFORE links between the I/O nodes to indicate its requirements (which are extracted from the program control structure).

It should also be noted that the THENs on higher nodes also serve a purpose during refinement. They are the official markers of what will happen before or after a node. Thus, any refinement of the node contingent on what has already happened in the program has this structure to examine. The point at which something happens in a program is important in determining where the data model should add nodes, in matching arguments to the nodes to supply them, in the evaluation of concepts, and in the determination of predicates.

The argument model is also responsible for making additions to the control structure. For example, when the argument handler discovers it needs a new node to provide an argument, the (MAKE (PREMETHOD ...)) goal is handled by MAKE-PREMETHOD, a module dispatched to by the planning routine. MAKE-PREMETHOD takes the goal of the following form:

```
[(MAKE (PREMETHOD A))
 TO: <- [B (-relation- A)]]
```

where relation is BEFORE, DURING, or AFTER. It makes a kind of METHOD called a PREMETHOD that has A and B in the prescribed relation (either [A, B], (AND A B), or [B, A]). This introduces the step B into the existing control structure. (The pre- indicates that the method comes before the actual refinement of A.) Besides the additions made by the PREMETHODs, there are also additions and modifications of the control structure made by the SCHEMA. The operation in a SCHEMA to change nodes is [(IMPLEMENT X) AS: <- Y], which turns the existing X nodes into Y nodes. The operation to add nodes is [(DO X) LOCATION: <- (BEFORE Y)], which corresponds to the MAKE PREMETHOD goal where X is the node to be added, Y defines the point in the program (a pattern), and BEFORE defines the relation.

The handling of IF-THENs during the analysis of a METHOD must determine what the control structure is going to look like. The first step is to test the predicate. This is done by asking WHETHER to determine the value of the predicate in the FUTURE. That is, what values could it have. The result is attached to the IF-THEN as a characteristic and the rest of the METHOD is analyzed accordingly. One possible outcome is that WHETHER can not determine the value until something else has taken place. If that happens, the analysis of the METHOD is suspended with a goal set up to restart it after the prerequisite has taken place. Finally, if the result of the WHETHER is EITHER the node to (TEST -the predicate-) is attached to the IF-THEN.

3.2 Iteration

Operations on the elements of a set or other object require the coordination of iteration and function. The functions to be performed are specified in terms of what should be done with each element of the set. The iteration methods specify how to get access to the contents of different parts of a data base or different kinds of Lisp structures. This would present a conflict if the programwriter were to refine the operation in the same order as the target language constructs that will implement it, because the functions to be performed inside the iteration are often specified at the high levels of the design hierarchy in terms of set elements and the iteration mechanisms are normally designed at the low levels of the design in terms of data bases and Lisp structures, while the control hierarchy in the Lisp constructs is just the opposite. To handle these potential conflicts the iteration control and the function are specified in parallel. The iteration control is specified with

[(ITERATE -object-) FOR: <- -element-]

where "object" is something with elements or something with a part that has elements and "element" is an element of the object or of some subpart of the object. For example, "object" may be (SET ...) and "element" the elements of the set. On the other hand, "object" may be DATA*1 and "element" may be ITEM*1 with several levels of structure including several levels of ELEMENTs in between.

As will be described with the data model, data bases have an internal hierarchical structure with parts called DATA-SECTs and RECORDs forming levels in the structure. The basic algorithm for the iteration through the elements of a data base is to provide an iteration level for each of these DATA-SECT or RECORD levels having multiple elements. The iterations are actually done at the Lisp level using DO loops.

The design of the iteration is responsible for the access code to the desired parts of the elements. For the parts of the contents of a datum, this means the appropriate part of the item structure. For part of the identifier, it will be a part further up in the data base hierarchy. If the only parts needed are parts of the identifier, it may not be necessary to include all of the levels of iteration. This means the cases must be characterized before the iteration methods are chosen to identify their requirements. This is handled by a procedure called by the INTENT for ITERATE which leaves the results as markers on the event. Looking at the iteration from the viewpoint of the purposes, the INTENT is simple. The first section of code done is the INIT, which is responsible for handling any initializations. The STEP is written in terms of the (ELEMENT *SET) and will be executed on each such element. The FINAL completes the iteration and is its result.

3.3 Providing Arguments

A major part of the work of the argument and control model is involved with making sure that arguments will exist for every node that needs arguments. This happens in three major ways. If the argument is available in the program, the correct node must be found and used. If it is not, the argument must be produced in some way. If there is any way that the designated node could fail to provide the desired argument, provision must be made to detect and handle that eventuality.

When the INTENT indicates that a concept is used as an ARGUMENT, it is necessary to determine whether it is available as an argument. The first check is to see if it is a constant, a primitive quantity, or an arithmetic expression. Obviously, if it is constant the value can be directly inserted as the argument. The only primitive quantity

in the system is TODAY, which requires the insertion of the CODE-METHOD for producing it, but is otherwise no more difficult that the insertion of a constant. If the argument is an arithmetic expression, it must be turned into a call to COMPUTE it. To do so, the concept (COMPUTE -the expression-) is formed. A goal is then added to make a PREMETHOD to include it in the control structure. This will produce the desired effect of analyzing the expression, because the step of the PREMETHOD will be analyzed and analyzing the COMPUTE concept will separate out the arguments to the first operation in the expression and the process will recurse. Notice that there must be communication among the different modules of the same model. The module to make a PREMETHOD is part of the argument model, but its function is part of the planning activity of the programwriter. Therefore, the information necessary for it to perform the desired job is provided via a goal.

Otherwise if the argument is none of these, it must have been produced previously in the program. Tracing back through the previous steps means going through the control structure of the METHOD. This is one place that the argument and control view of the program is important. It could be arbitrarily difficult to determine for certain whether such an argument would always be available if, for example its calculation were one part of a conditional. Such situations do not come up in practice because the methods are written to aid the programming. If it were necessary to make a method construction that would be difficult to analyze, helpful assertions could be placed on the constructs. This is consistent with the way people deal with confusing program structures -- remembering their important properties rather than figuring them out each time. In the normal situation there is a single path back through the control structure as in figure 15 (from D to C to B to A), which may have to go through the refinement structure and into conditionals, but without difficulty. The search does not attempt to go

down into refinements, such as those below B -- they may not exist or they may even change later.

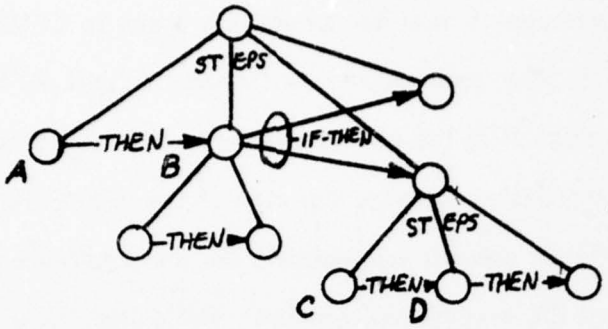

Figure 15. Searching for arguments

Going back through the THEN structure, there are several situations which may occur. If the item is returned by a node, then it should be used as the argument. This is indicated by the RETURNs in the INTENTs for the nodes. To use it the argument concept is specialized by the concept (RESULT -node returning it-). If something is returned by a node that the argument is a part of, then the argument is the appropriate part of the concept returned. This happens because the search goes up through the refinement structure. For example, in the program for accepting deposits (in the introduction and more fully in the scenarios following), the two main steps were to (ASK A-DEPOSIT*1) and then to (ASSERT A-DEPOSIT*1). At some point in the refinements of (ASSERT A-DEPOSIT*1), the actual amount of the deposit will be needed. (AMOUNT-M A-DEPOSIT*1) will be provided by one of steps of the refinements of (ASK A-DEPOSIT*1), because (ASK A-DEPOSIT*1) claims to return the whole datum. Therefore, the concept generated by the evaluator for the argument is

(AMOUNT-M (A-DEPOSIT*1 (RESULT (ASK A-DEPOSIT*1)))).

It is necessary to use the (ASK A-DEPOSIT*1) node because the node structure below it

may change through modification of the program leaving a pointer to a node that is not in the program. More commonly the lower nodes will not exist yet and waiting for them could get the goal structure into an impossible state. It is sufficient because the data model is charged with the responsibility to make sure that such parts are marked as needed. To actually resolve which piece of code returns the argument is handled in the coder.

Often there is no mention of the desired concept along the control structure from the node needing an argument back to the beginning of the program. In such a case a new step must be added to produce the argument. The analysis module does not actually change the program structure itself, it adds a goal to make a PREMETHOD to (PRODUCE -argument-) BEFORE the place where it is needed.

The new node to PRODUCE the argument does not say how it should be done. That is left to analysis and METHOD selection. The first step is to check whether anything is known about the argument. The node that wants the argument may have a KNOW property that provides useful information that will assist in producing it. The data model (and any of the considerations made for the data model) has a more general notion of the "node wanting the argument". If an arithmetic computation is to be performed, such as (+ (* A B) C), then the desired meaning of *argument* is the ultimate arguments, i.e., A, B, and C, rather than (* A B) and C. These are known as the ULTIMATE-ARGs of the computation. Because of this, information about A could be on the whole computation. Therefore, each node that is an intermediate computation for a higher node must be checked. The KNOWs that are used provide information limiting the arguments. For example, a KNOW might limit the contents of a SET, or specify that only certain arguments are to be used.

Once any KNOW properties have been handled, three possibilities exist (assuming the need for the argument was not eliminated): it has a source in some other program execution (i.e., some other ACT) and must be stored by that program and retrieved to produce the argument, it has some definition that can be turned into a method for computing it, or the user must be asked for the value. The user only needs to be asked when the argument is a part of the specification of the program represented by a dummy. In that case the actual desired value for the dummy must be determined during the running of the program (e.g., an identifier for the particular datum the user wants stated). Of the other two possibilities, either or both can be true. If both are true, it will be up to the data model to decide which is valid in the situation once the definition has been analyzed. If there is a definition, it must be expanded and analyzed. Since the data model can not decide whether the definition is to be used until after it has been analyzed, it is not included in the control structure until the analysis is complete.

After the analysis of the PRODUCE node is complete, a METHOD must be chosen. There are different METHODs for PRODUCE depending on the existence of the definition or a source. If there is only a definition, the METHOD is to use it as analyzed. If there is only a source, the METHOD is to RETRIEVE it. If both exist the METHOD depends on the data model's assessment of which to use, possibly including a conditional to determine which to use at run time.

3.4 Failure

Even if there is a node that will provide the desired argument, it is possible that the node might fail. There are only three situations in the programwriter when

failure is a possibility: retrieval, asking, and the passing of a required predicate. The last two are not a problem, because they are handled by doing an error exit from the program. Retrieval is a more interesting problem which requires the cooperation of the data model.

The handling of failure in the programwriter is similar to the handling of IF-THENs, only the condition is not explicit and it is possible that there is something that can be done to avoid the situation. The issues about failure that must be handled include whether something can fail, how to test whether it has, and what to do if it does. The test for whether something can fail is handled by WHETHER. Testing whether failure has occurred is sometimes provided in the METHODs by constructs of the form (IF-THEN P FAILURE), where P is a predicate and FAILURE is a concept indicating that the METHOD has failed. Otherwise a call can be generated of the form (TEST (IF-FAIL -node-)) and the selection of a METHOD is handled by SELECT-METHOD. What to do if it does fail depends on what IF-FAIL properties exist on the node and the superior nodes in the tree and what other options might be possible.

Consider the problems that arise in retrieving a datum from a data base. The datum goes through various stages of refinement during the design process (to be discussed in detail in the next section). At the top level, the datum is considered abstractly. If there is a defined value for it, it has that value. At a more concrete level, the datum may have a representation in a data base. Then it only has a value if some program has already stored the value in the data base. There may be a legitimate value for the datum, but if it has not been stored the attempt to retrieve it will fail. In considering METHODs at a high level in the design tree, the production of the value represented by the datum is not considered to fail. At a lower level, the control

structure must be provided to handle the failure to retrieve it. For reasons such as this, the error handling structure goes through refinement right along with the rest of the program structure.

The first problem for the failure handler is deciding whether failure is possible. Ultimately, the only kinds of activity that can are kinds of RETRIEVE, because the failure of anything else in this programming domain is the result of a lower level RETRIEVE or an explicit call to failure. The failure analysis routine in ANALYZE knows this and does further checking for any kind of RETRIEVE. That is, it asks WHETHER to test (RETRIEVE ...). Remember, WHETHER can return TRUE, FALSE, EITHER, or wait for some specific information.

Some of the more interesting situations that the WHETHER module will have to handle are discussed in chapter five, after a set of examples have been explored and information from the other models is available, but consider the basic situations: A RETRIEVE call can not fail if the object exists. If the object was initialized, it must be there. (There are no cases of removing things that existed initially.) If the object is being retrieved from a structured container (one that always has its parts if it exists), it will not fail. If the call failed (or succeeded) previously on the control path, it will still fail (or succeed). If a refinement of the call can fail, the call can fail. And finally, if the object might not exist (the data model will help out with characterizations), the RETRIEVE might fail.

As with IF-THEN, failure only turns into branches in the control structure if either failure or success is possible. If success is the only possibility, the failure point is ignored. If failure is the only possibility, not only is the failure branch appropriate, but there is no reason to include the node that will fail. For this reason, even if WHETHER

does not know the answer but can eliminate FALSE, it will indicate that with the DON'T-KNOW reason.

Once it has been decided that failure is possible, the next problem is how to detect it. Figure 16 is the algorithm to handle detection:

```
1) does it produce an error upon failure?
yes: 2) is there a default value?
     yes: 3) is datum initializable?
          yes: 4) make initialization routine
          no:  5) make test procedure and substitute default value
     no:  6) make test procedure and failure action
no:  7) is result ambiguous?
     yes:  8) is result correct?
          yes: 9) use it, no failure
          no:  10) is datum initializable?
               yes: 11) make initialization routine
               no:  12) reject implementation of container or value
     no:  13) is there a default value?
          yes: 14) is datum initializable?
               yes: 15) make initialization routine
               no:  16) test result and substitute default value
          no:  17) test result and do failure action
```

Figure 16. Failure algorithm

Does it produce an error upon failure? There are two ways that operations fail. They can either return some value indicating that they have failed. Or, they can produce an error -- a situation to be avoided. The Lisp model will know the answer. Is there a default value? If there is a legitimate value for the object when the retrieval fails, then maybe it can be used. Is datum initializable? Even if the value is known, it may not be possible to plug it in ahead of time. The problem is that it may not be able to create a place to put the value without knowing ahead of time all possible data that might be received -- more about that in the data model. If it can be initialized, go ahead and make an initialization routine. Making a test procedure means a test whether the

operation will fail. That is, (TEST (IF-FAIL (RETRIEVE ...))). If there is a legitimate value, the failure branch can be for the result of the RETRIEVE to BECOME that value.

Is the result ambiguous? If the failure result returned by the RETRIEVE is in the range of legitimate values it can return, then it is not much good for indicating an error. Is result correct? However, if it the value that should be returned anyway, that is fine -- forget about the failure. If the failure value is not distinguishable from a legitimate value and the value can not be initialized, then the only option is to reject the implementation of the representation of the value. Testing the result means generating a test that compares the result to the failure indication value.

Once failure has been detected, handling it is fairly simple. In the case of RETRIEVE it usually just means using the default value instead, as mentioned. The default value can be found on an IF-FAIL for the call to retrieve or a superior call, or it is available as a property of the implementation. In cases where there is a failure action to be taken, that is found on an IF-FAIL on one of the superior nodes. All very straightforward except that there may be default values or failure actions on more than one of the superior nodes. The programwriter uses the first one it finds going up the nodes. This always works, but there are other alternatives that will be explored in chapter five.

3.5 Interfaces with Other Models

Taking another look at what the argument and control model does, consider how it relates to the other models. It provides interfaces of both the active and passive kinds. The three major modules, the argument handler, the failure handler, and the PREMETHOD maker are mainly interfaces with the design model. That is, they are used

in response to the needs of the refinement objectives. The other active services provided are through EVALUATE and WHETHER. In general, any kind of searches through the control structure needed to answer a question is handled by routines provided by the argument model.

The terminology of the argument model is used by all of the other models except the domain model. The target language model needs all of the control concepts to determine what kind of relationship should exist between the primitive operations. It uses the IF-THENs and IF-FAILs to indicate the need for branching constructs, the ITERATEs to indicate the need for iteration constructs, the THENs and ANDs to determine what constraints are on the order of the constructs, and RESULT to indicate where the arguments come from.

The argument model and the data model have a close relationship. The data model makes requests in the form of properties left on the node claiming to be a SOURCE or USE to include a new node in the tree BEFORE, DURING, or AFTER the indicated node. The argument model picks these up while handling the BECOME below the node and includes them in the structure with the appropriate relationship to the BECOME. The argument model makes requests to the data model by setting up goals to CHARACTERIZE the data and by including calls to PRODUCE pieces of data. The argument model also maintains the ULTIMATE-ARG characteristics to aid the data model in finding sources for data.

The I/O model also must rely on the argument model. It has a set of refinements for I/O operations, which it must rely on the argument model to handle. The refinements use the same concepts as other METHODs, so to the argument model this is no different from any other refinement.

Section 4 The Data Model

The programming done by the programwriter is in a sense data driven. The connections between the programs it writes are in terms of the data they need from each other; the actual programming constructs used depend on the form of the data; and the decisions about efficiency are primarily concerned with the effects of program changes on the data. Hence, much of the effort of the programwriter is concerned with handling the data. Information about the data must be collected; it must be turned into an appropriate description of the data; and the data must be implemented to optimize its use. Data goes through a hierarchy of descriptions. At the most abstract level, a piece of data such as an A-DEPOSIT is a single entity. It is produced by one program and used by another. Looking more closely at it, there are two main parts: the identifier, specifying which A-DEPOSIT it is; and the contents. These in turn may have parts. In another sense the A-DEPOSIT used in the program is a further abstraction of the A-DEPOSIT representation abstracted in the domain model from the DEPOSIT activity representation, which was in turn abstracted from knowledge about making deposits in the real world. Exactly what is recorded in the A-DEPOSIT depends on what is needed by the different programs. Once the parts of the data item have been determined, they take on the aspects of data themselves. The later refinements of the program that supply these parts and use them need to know their nature in the same way that the higher level steps need the whole datum. In another dimension the data is first viewed abstractly in terms of sets without concern for how it will be stored or retrieved. Then it is viewed in terms of the data bases necessary to store it, the preparation of the data for storage, and the operations for storage, retrieval, and maintenance of the data base representations. Finally, the data must be viewed in terms of the Lisp structures that the ultimate programs will use.

The data model views the set of programs as creating and using data. The important facts about a program are what kinds of data it needs, what kinds of data it produces, and what changes it might make to data. The data model is the only model that has a view more global than the individual program. (Even in the argument model when considering the possibility of initialization routines to avoid failure, that was really the interface with the data model.) The reason for the global view is the nature of data. Data has an existence independent of the programs. In fact, remembering from the domain model, it is the programs that exist to support the data rather than the other way around. The data model is interested in preserving the consistency of the data by insuring that all of the program parts exist that are necessary to keep them intact. That means the programs must be able to handle any situation that is possible for the data. To see what this entails, it is first necessary to have a model for data. This one is a little more complicated than the one in the introduction.

A datum has a history relative to the execution of the programs. A program brings the datum into existence, which may cause one of two things: First, it may create a datum that did not exist before. Second, it may change an existing datum. (The distinction will be important when designing data bases.) The program may get the datum either by inputting it from the user or by computing it from other data. Also, a datum is *used* by a program either to output to the user or as an argument to a computation.

Notice that this view differs from the domain view of data, that it exists at the time of the real world event, whether or not the program produces a representation of it at the same time.

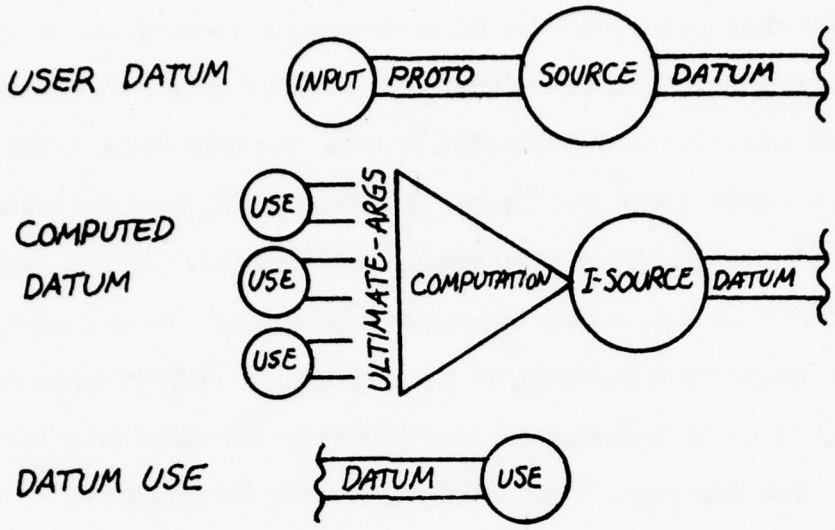

Figure 17. Datum sources and uses

To handle this view properly in the programwriter, there must be terminology to label the important parts. The terminology will make it possible to specify the state of information and to determine locations for the code the model may require. Consider first inputting a datum from the user. The datum enters the program through a port called INPUT into a stage called PROTO. In this stage the parts are being gathered but it has not yet been accepted as a legitimate datum. PROTO is used to specialize a concept refering to the proto stage of a datum. For example, the ASK method, used to get the A-DEPOSIT from the user, returns the (*datum* PROTO), where *datum* is what the user was asked to input. While in the proto stage the objects being accumulated must pass any tests required of a suitable datum before it can pass through the SOURCE port. Once it passes through that port it is an accepted datum. Before that point it can not be used for any purpose but its own verification. Passing through that port is marked in a METHOD by BECOME of the datum. For computed data the situation is similar. The

computation, as far as the data model is concerned, is a black box. Going into it are a set of ULTIMATE-ARGs. The result, after passing any required tests goes through the port called I-SOURCE, (for immediate source) which occurs at the BECOME in the METHOD. The reason there are different ports for the two forms of creation is the different stature of an input datum and a computed datum. The user is considered to be the ultimate source of data. If a datum is computed then its ultimate arguments come from somewhere. At some point every datum has SOURCEs, perhaps through several stages of I-SOURCEs. This observation is the basis for the view of using a computed datum versus the same datum from the user. The question is which one has the most recent SOURCE. If the computed version has more than one source, it is the most recent that is considered.

The other end of the datum's history is its USEs. These are declared by the INTENTs for RETRIEVE. If the datum is going to be used in a computation, it will be an ULTIMATE-ARG of something. If it is also going to be output to the user, this is indicated as the OUTPUT in an INTENT. For the uses to take place, the integrity of the data must be preserved from the source (SOURCE or I-SOURCE) to the use. If both are within the same program execution there is no problem. The difficulty and the primary chore of the data model is in preserving the data between executions. This calls for an external form of storage. This is also where the *set* of programs comes into play. The variations in the contents of the external storage depends not on the alternative paths through a program, but on the alternate program execution histories.

Figure 18. Basic storage model

The basic view of external storage at this level is an amorphous object into which data is ASSERTed and from which it is RETRIEVEd. This is the level at which the argument model interfaces with the data model. Once it is known that data must be RETRIEVEd, there is a need for storage. In fact only then is there a need for the source. The data model has control over what happens at the source. When it determines that a datum will be RETRIEVEd, it attaches a requirement that it be ASSERTed.

Once needs have developed for data in the programming situation it is necessary to characterize the data. Characterizing means finding the properties of the data that are important to the data model, such as the sources, the uses, the parts of it that are needed, when it is defined, how many there might be, and whether they change. This is necessary primarily to provide a better view of what the storage must look like. Beyond that, the way the programs must react to the data and therefore their structure is dependent on the major properties of it. To characterize a datum it is necessary that everything that might effect the characterization has already been fully explored. In other words it is necessary to have a closed domain of knowledge about the uses and sources of the datum. This is a result of the view of the model, but it is conflict with the refinement structure and the possible variation in refinement from one part of the program design tree to another. The way this is handled is by going down the goal list

and checking for goals to expand or analyze that involve the datum. If there are any, even if they are not active because they are waiting on some other goal, the characterization must be suspended. This situation does not normally happen because the characterization goals are taken care of after the expand and analyze goals. Next, the SOURCEs and USEs of the datum are determined. These are used in the rest of the characterization and much of the later planning.

The SOURCEs are the easiest to determine. Part of the job of the INTENTs used in analysis is to specify when the call is a SOURCE for a datum. That takes care of data input from the user. The rest is determined from a DEFINITION, and the expansion and analysis of a DEFINITION also declares ULTIMATE-ARGs which in turn must have the SOURCEs. Determining the USEs of a datum happens in two parts. First, the regular USEs are determined, then during the testing for CHANGE the SOURCEs that will use the datum to change it are added. The regular USEs are those in which the datum is needed by some other program. These are marked by the INTENTs in the same way that the SOURCEs are. Since RETRIEVE is the only way of obtaining a datum outside of the program the various retrieve INTENTs are the only ones to do this.

There are three major features about which the data is characterized. The first is whether or not it changes value. If it changes value, it is necessary to plan in the implementation of storage for the ability to change the value. If it were in a data base it would have to be uniquely keyed to even find it to change it. The second is whether or not it is initially defined. That is, before the program that is its source has taken place, does the datum have any meaning? Finally, is the datum known to the system before its source has taken place. That is, does the system know that this particular datum will come into existence? What knowing a datum boils down to is that the identifier set for

the data is constant. If a datum is known, a specific place can be planned to store it. If a datum is not known, a dynamic data structure must be planned that will expand to handle the particular data items that occur.

Determining whether a datum can change is an interesting problem. Like many of the other problems encountered by the programwriter, there are fuzzy areas where any particular algorithm might not be good enough to tell. This is not a great problem however, because it can always be safe by erroring on the side of change. The general test is to propose the existence of some datum with a pattern of identifier and content values just sufficiently constrained for it to have been accepted at some point in the past, then see if it would be accepted by any of its sources as a datum now. Pragmatically, most data either can change or part of its identifier is the time of execution of the source, and therefore can not change (only one execution allowed per point in time). When it is determined that a datum can change, the source responsible for that change is added to the list of uses.

There is one additional special case for changes. It occasionally happens that a program's ability to change a datum is restricted to some time period. This is desirable for more reasonable data storage requirements by making possible the deletion of old information. The ability to make changes can be tied to either time or events. For example, a program to correct deposits might have the requirement that the ETIME be less than a month out of date or that it be since the last stating of the balance. In either case the requirement must become a property of the CHANGE characterization. That way the information can be used both to design the form of the storage and establish bounds on its use.

Determining whether the datum is defined involves the properties of the

datum and the patterns of usage. The question is really whether the datum can ever be needed and not be defined. At the set level, once something is defined it remains defined, so it is only necessary to determine whether the datum will have a value before it can be used. However, at the data base level and the Lisp level data can be destroyed. If it were accessed after it were destroyed, the situation would again be one of being undefined. This can only happen purposefully because the information would never be declared useless if it could be accessed by one of the programs. The kind of situation where this might come up is when the only use of a piece of data would be to reject something that could be rejected on other grounds.

Data that is defined is marked in some way. Specific entities that have an initial value have an INITIAL specialization in the knowledge base with a specific value. Other data is defined by virtue of the kind of data it is. Sets always are defined with a default value of empty. Because of that the concept of a count, which refers to a set, has a default value of zero. The concept is given the characteristic DEFINED.

Whether a datum is known involves an analysis of the identifiers. If the identifier set of the datum can change, then the datum is not known. Pragmatically, it is a little more restrictive than this. It does no good for the identifier set to be constant but infinite, or constant but with unknown values. The identifier set must be given as an explicit set for the datum to be known. That means specific slots for the values could be anticipated and provided by an initialization program. If the data is known, it is given the characteristic KNOWN.

Data also must have its contents and needed identifiers determined during characterization. The contents refer to the parts of the datum that are used. Determining them just means looking at the form of the uses. Only those parts that are

used are needed. It may seem that the identifiers will always be needed, but if the data is used in whole sets (for iteration), it may be that some of the identifiers are not actually used. If the datum has been characterized as changing, all of the identifiers will be needed to make the changes possible. It is also possible that a datum will have no contents. This happens when the only uses of the datum are based on its existence. Once the needed parts are found, they are given the characteristic NEEDED to help out the I/O model.

Sets must also receive some extra attention. The most important feature of a set is its size. Not only is it important to know how a set grows to decide between alternative designs, but it is also important to recognize sets with a fixed number of elements, particularly zero or one, to simplify the original code. The programwriter can recognize the important circumstances when this can happen, although it involves some work as will be seen in the scenario. In cases where more than one set with non empty intersections is used the union of the sets must be characterized, because that is what will be implemented.

Characterization for sets also has the responsibility of declaring special properties effecting the requirements of the set. One such special property will be seen in the scenario, when a set of data restricted to be SINCE an event can be implemented without requiring the ETIME, by carefully keeping the data base representation pruned to just the set. Thus, it is possible for the characterization of the data to add a requirement that the data base for it be PRUNEd.

Once the characterization has taken place, and the program structure has been further expanded on the basis of the characterizations, it is time to consider an external representation for the data.

4.1 Data Bases

The idea of a data base is a handy abstraction for pinning down the organization of data without having to specify the programming language constructs that will represent it before the programmer is ready. For the data model, it is the first refinement of the storage idea. There is no absolute requirement that the data be considered in terms of data bases. Lisp implementations could be proposed directly from the requirements already known, but the task of selecting the appropriate implementation would be more complicated. Organizing and exploring the requirements and properties of the data in terms of data bases provides the information that makes the Lisp implementation an understandable process.

Data base is used as an abstraction for any kind of data structure that holds information between program executions. A data base represents either a datum or a set of data. It can be viewed as a container for storing the data, having an existence independent of the data. If the programs use different sets with non-empty intersections, the data base should represent the union implying that one or more of the sets will not be specifically represented. In either case the programs that retrieve from the data base must take care of such discrepancies by filtering the data. Furthermore, it may not prove to be efficient for the data base handling programs to have data base represent exactly the set desired for the set level operations. Thus, in designing a data base, the desires of the set level operations are taken into consideration, but the resulting data base may represent a superset of the desired set.

The representation for a data base forces a certain amount of structure onto the data. There are several basic categories that the data is divided into, depending on

whether there is one datum or more than datum, whether there are identifiers that will be used in the accessing of the data and how many, and whether there can be more than one datum with the same identifier. The identifiers, if there are any will also be put into an order to provide a hierarchical structuring of the data.

The data base model for the programwriter has been chosen because it is simple and flexible. It is not the only possible model, but it works nicely for the domains of interest. The primary advantage is that it is a fixed part of the programwriter. It simplifies design by providing a framework to work around[4]. The model for data bases used by the programwriter is as follows:

1)DATA-BASE is an entity by itself, so it has a NAME. All of the divisions of the contents are PARTs of the DATA-BASE. The one with all of the contents is called the DATA.

2)DATA has all of the entries in it. It comes in one of two forms: RECORD or DATA-SECT, to which it is said to be EQUIVALENT.

3)RECORD contains zero or more ITEMs all of which need not be distinguished. That is, the programs do not need to distinguish among them, and therefore they are represented in the data base in a way that does not provide for distinguishing them. If the ITEMs all need to be distinguished, the RECORD is EQUIVALENT to the ITEM.

4)ITEM represents one piece of information. That is, the contents of each datum is an ITEM. Since the contents can have parts, the ITEMs can be further structured.

5)KEY-LIST is a list of the parts of the identifier needed to distinguish the data sufficiently. That is, a list of concepts identifying the classes of the necessary identifier parts. These keys determine the subdivision of the DATA. If the KEY-LIST is not empty, the DATA is EQUIVALENT to a DATA-SECT.

6)DATA-SECT is a data base part consisting of any number of KEY-DATA-SECTs.

7)KEY-DATA-SECTs are in turn, pairs of KEY and either another

[4]That might be restated in the terms of Mark [Mark 1976]. The programwriter has a model of data bases about which it is expert. It can then formulate its problem in terms of that model and is then able to solve the problem.

level of DATA-SECT or RECORD. A property of the DATA-SECT is the element of the KEY-LIST representing the kind of KEY that partitions it. The KEY-DATA-SECTs have a specific KEY of the class indicated by the DATA-SECT above and either a DATA-SECT with another element of the KEY-LIST or a RECORD if all of the elements of the KEY-LIST have associated DATA-SECTs.

With this structure it is possible to have a useful variety of data bases. The simplest data base would be one for a single datum, with the following structure:

```
DATA
EQUIVALENT RECORD
        EQUIVALENT ITEM
```

A more complicated data base for data with a single identifier of ETIME that distinguishes each ITEM would have the following structure:

```
DATA
EQUIVALENT DATA-SECT
        ELEMENT KEY-DATA-SECT
              FIRST-PART [KEY ETIME]
              SECOND-PART RECORD
                      EQUIVALENT ITEM
```

To produce a data base structure for data, a goal is set up to IMPLEMENT it. Implementing a data base consists of deciding what this structure should be, instantiating it for the data, and making any other additions or modifications of the program structure that are called for to preserve the integrity of the data base. Deciding what the structure should be is driven by the use of the data. The basic commands that need to be considered are ASSERT, RETRIEVE, and PRUNE. Obviously, the way RETRIEVE is used is going to determine what parts of the identifier have to be included as KEYs, but ASSERT also can cause a need for a KEY. This happens if ASSERT can be used to update existing items in the data base -- the reason for the CHANGE characterization. In that case it is necessary to have each item fully keyed to even tell if the exact item exists to update. The KEY-LIST is determined by the subset of the identifiers that are used to

RETRIEVE or ASSERT with updating. The PRUNEing activity is not needed to determine the elements of the KEY-LIST, but it may be used in determining the order. Pruning is removing from the data base items that can no longer be used. If they can no longer be used there must already exist a usage that distinguishes between those and the rest, so any identifier parts needed for pruning must already be needed for something else. The algorithm for ordering the keylist is: 1) Get the list of identifiers used from each use of the data. 2) Order the identifiers by number of uses, with the most used identifier first. If there is an order such that all of the uses will have the identifiers they need first, this will find it. 3) If it is not true that all of the uses have their identifiers first, the module resolves the problem in favor of the programs that are used more.

Once the key list has been determined, the determination of the data base structure is fairly simple. If the KEY-LIST is the whole identifier, then the RECORDs will be EQUIVALENT to the ITEMs. Otherwise, the RECORDs will contain an arbitrary number of ITEMs. If the KEY-LIST is empty, the DATA is EQUIVALENT to the RECORD. After this, each KEY corresponds to a DATA-SECT level. The actual implementation takes place by applying the SCHEMAs that match to the data that is to be implemented. This is a situation where more than one SCHEMA is involved. The original SCHEMA that calls for the KEY-LIST ordering must call on other SCHEMA in accordance with the results.

At this point in the refinement by the data model, there are calls for RETRIEVE, ASSERT, and PRUNE (plus the other parts of the data production and usage -- input, computation, and output) and a DATA-BASE to put the data in. From here the process is to further refine the operations to take into account the parts of the DATA-BASE and their properties. To handle the operations at this level, the model needs two more ACTIVITYs: INSERT and CREATE. These five ACTIVITYs are used at all levels of

the structure hierarchy in the data base to manipulate the parts. RETRIEVE gets the desired part of the data base. It is used not only to implement set retrieval, but also in the METHODs of other data base procedures to retrieve the parts they need. ASSERT makes something into a specified part of the DATA-BASE, which may include making missing parts of the DATA-BASE structure. PRUNE filters the unneeded items from the DATA-BASE and also is used to make the contents of a DATA-BASE correspond to a set dependent on time or events, as mentioned in the previous section. INSERT is just a special case of ASSERT for the case where the part is being added to a variable length structure (a structure with ELEMENTs). By implication an INSERT never changes an existing part. CREATE is used to produce the required DATA-BASE substructures that do not yet exist. It is called to do initialization, to build new DATA-BASE parts for PRUNE, and to produce the missing parts when RETRIEVE inside of an ASSERT fails.

There are METHODs for each of the ACTIVITYs which provide the refinement of the data base handling. These ACTIVITYs all approach the data base as a whole. For example, (RETRIEVE KEY-DATA-SECT) retrieves from the DATA-BASE. The refining of these operations turns them into a lower level kind of operation, known as Lisp level operations. These operate in terms of the containing part of the data base (the SUPERPART) or the parts contained (the SUBPART). (To give these refining METHODs the handles they need on the DATA-BASE structure the terms SUPERPART and SUBPART are interpreted by an evaluation module to climb up and down the DATA-BASE structure ignoring EQUIVALENT parts.) For example, (L-RETRIEVE KEY-DATA-SECT) retrieves the KEY-DATA-SECT from the DATA-SECT it is in. The refinement of (RETRIEVE KEY-DATA-SECT) for the second DATA-BASE example above would be:

```
<1>[(L-RETRIEVE DATA) FROM: <- DATA-BASE],
<2>[(L-RETRIEVE KEY-DATA-SECT) FROM: <-
                        (DATA-SECT (RESULT <1>))]
```

Notice that the DATA is EQUIVALENT to the DATA-SECT, so the DATA that is retrieved in <1> *is* the DATA-SECT needed in <2>.

Once the data model gets the operation to this level, it is finished. The refining has been carried to the lowest level of the data formalism. The rest of the problem is up to the target language model. It must produce constructs which will carry out these operations.

4.2 Interfaces with Other Models

The data model has been used for four main purposes: to label the parts of the program with respect to their part in the production and use of data; to characterize the data with a view toward implementing it, including determining what will be required of the programs for storage; to design data bases to store it; and to refine the operations to take into account the structure of the data bases. These also show how the data model interfaces with the other models.

The data model owes the form of the DATUM concepts to the domain model. It used the basic form as a base from which to refine the data representations needed in the program and also used the properties of the DATUM to determine the DEFINED and KNOWN characteristics.

The relationship between the data model and the argument model is two way. The data model characterizes for the argument model, providing it with facts needed by the argument handler and failure handler. That includes the facts needed to determine whether PRODUCE should be turned into a COMPUTE, a RETRIEVE, an ASK, a system call, or a constant. The data model also provides routines that answer some of the argument

models questions, such as "Is it initializable?". The argument model provides the capability of adding new operations at the SOURCEs, in fact it provides the ability to exactly find these locations. The data model knows these locations because of the markers left by the INTENTs, but the argument model can find the BECOMEs where they take place. This also means that such operations as finding out what happens SINCE an event (e.g., evaluating the size of a set occuring since an event) is a joint effort on the part of the two models. The data model is responsible for what happens between programs and the argument model is responsible for what happens during a program relative to the exact point of the event.

The interface between the data model and the I/O model is the terms INPUT, PROTO, and OUTPUT. INPUT and OUTPUT are used by the data model to classify the data. INPUT means that it comes from the user, and therefore takes priority over other forms of the same data. OUTPUT means that it is going to the user rather than as an argument for some other data. PROTO is used by the I/O model as an indication that the data is still safe to modify or reject.

The interface with the target language model is provided by the data representation concepts and low level operations they share. For example, the data model looks on KEY-DATA-SECT as an element of a DATA-SECT and as having a KEY and a DATA-SECT or RECORD. The target language model looks on this same concept as something that has two parts, one of which may be used as an index for retrieval.

Section 5 The I/O Model

One of the fundamental operations in any system of programs is taking

information from some source outside the bounds over which the programmer has complete control. This occurs in a large system whenever information is passed between separate modules and in small systems when information is read from an outside source. There are two basic problems involved, forming the appropriate kind of request to get the information, and verifying the result by testing to see that it satisfies any requirements known about the information. This second step is necessary because the programmer does not have complete control over what might be returned. The converse operation of transfering information to some outside destination is also important and presents many of the same features. This might appear easier since the program has control over the data, but the program must still observe the conventions of the external environment and present a consistent image. The key to both of these problems is that the behavior of the program with respect to the external agent be understandable.

The example of external interaction that the programwriter has to handle is simple I/O with the user. This I/O is of three basic types: requests to the user for data, statements of data to the user which were requested by him, and statements announcing error conditions. The programwriter uses a model of I/O interaction to provide for the refinement of these three types of situations.

The I/O model emphasizes the need for consistency and understandability with respect to the user by viewing a program as communicating with the user through the console. From this point of view, the program consists of a sequence of exchanges between the user and the program. The only concern of the model is how these interactions takes place. Questions to the user must be composed to make the desired response unambiguous without the statement of the question being redundant. There

must be sufficient ordering constraints to prevent unnecessary interchanges and preserve the coherence of individual interchanges in any possible sequence. Individual interchanges need to be composed in a pleasing way, starting on a new line, prompting the user when input is needed, and maintaining consistent spacing.

Part of the boundaries of the model are set by the target language. The I/O is conducted on a standard console, using the normal subroutines provided by Lisp: READ, PRINT, and PRIN1 (print the argument without preceding carriage return or following space). If the details of the low level I/O such as rubout processing, printing of individual letters, and interpretation of numbers were not taken handled by these subroutines, they would have to be coordinated by the I/O model. Given the subroutines of Lisp, the model need not include that level of detail. The model assumes that the primitive object for printing is the atom, and a whole object is read with one request. Leaving these details to Lisp also means that dates will have to be typed as numbers, rather than a more familiar format, but this is a detail that could easily be fixed by a canned subroutine. Using the Lisp subroutines still leaves such things as placement of carriage returns and spaces to the model.

The points of entrance for the I/O model are the calls for the three types of I/O. Requests to the user for data are specified by an ASK call. Statements to the user are specified by a STATE call. Both of these kinds of calls can be for either a DATUM or a PROPERTY of a DATUM. Error conditions are situations recognized by the failure analysis module for which there is not way to handle the situation within the program. That is, there is no IF-FAIL property on any of the steps of which the failing step is a refinement. The response by the programwriter is to design a failure return from the program with an appropriate message.

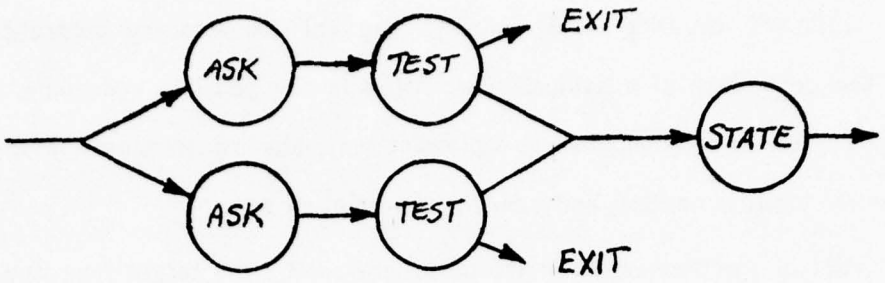

Figure 19. I/O view of a program

From these entry points the I/O model takes over by handling the refinement of the ASK and STATE calls and by providing the error messages for the failure returns. The basic view of the program by the I/O model is as a set of *I/O units* within the partially ordered control structure provided by the argument and control model. An I/O unit is a complete statement or request-response pair, normally taking up one line. Asking for the value of a specific property would take an I/O unit. To do an ASK or a STATE of a datum may take several I/O units, depending on the number of identifiers and contents. The model is concerned with the designating the correct I/O units for the program, with their ordering, and with designing the I/O units. The refinement of ASK and STATE can be divided into two parts. The first part designates the needed I/O units and partially provides their ordering. The second part designs the I/O units. Finally, there is part of the coding process that determines the rest of the ordering. The processing of failure returns is much simpler. The messages already exist in the knowledge base as the FAIL-MESSAGE property of the test to determine whether such a failure has occurred. Thus, the message produced when the test for a number fails is:

(FAIL-MESSAGE (TEST (IF-FAIL NUMBER))) <- ERROR_NOT_NUMBER

In the program the Lisp symbol corresponding to this concept will be returned as an exit from the program. This point in the program is still considered I/O and will be used in the coding stage to determine the ordering.

Consider the first stage of refinement for ASK. What it means to ask the user for a datum depends on the situation. In the simplest case where there is a single item with a fixed identifier, it is only necessary to print out the identifier as a description of what is needed and handle the user's reply -- one I/O unit. In the more typical case the request is for an element of a set whose elements must be distinguished in some way. In such a case it is necessary to first determine the needed parts of the identifier and then the contents. That is, first determine which element is being input and then its values. It is not always necessary to ask for all of the parts of the datum. They may be internally computable or just not needed. The design of the code for asking for a datum takes place after it has been characterized by the data model. The INTENT for ask waits for the characterization before METHOD selection to make sure there is complete knowledge about what parts of the datum are needed by the programs before designing code to get them.

The top level algorithm for asking for a DATUM is to ask for the identifier if it is NEEDED and then ask for the contents if they are NEEDED. If there is more than one part to either of these, there is a METHOD to ask for multiple parts by just asking for each without a prefered order. NEEDED is a concept of both the data model and the I/O model, providing for communication between the two. It may seem like the content will always be needed, but if the only content is the existence of the DATUM it will not be. That is, existence does not require any input from the user besides that already provided by the identifier. At this point, the problem has been broken down into asking for the specific needed parts of the datum. There is one case that can be handled before turning the rest over to the design of I/O units. If the needed property is the ETIME and time of execution of the program is the same as the ETIME, the value can be provided directly by the Lisp subroutine TODAY. This is an example of a part of a

datum that can be determined without resorting to the user -- the only one in this domain.

The top level algorithm for STATE is very similar. The datum must be identified before its contents can be printed. The identifier may have zero, one, or more parts and the contents may have one or more parts. If either the identifier or the contents have multiple parts, the order does not matter.

At this point the calls are of the form (ASK (-property--datum-)) and (STATE (-property- -datum-)). Each of these handles an I/O unit. The I/O model has two rules about I/O calls at this level that will be applied by the augmented METHOD selection procedure for these activities. They are: don't state something just asked, and don't ask twice. If a node to STATE a property needs a METHOD and there is a previous ASK node for the same property, the goal is cancelled. If a more than one node ASKing for the same property exists, they all use the same result, as in the RETRIEVE or COMPUTE case. The METHOD for ASK is as follows:

```
[(METHOD (ASK PROPERTY))
 OBJECT: <- (PROPERTY: DATUM):
 STEPS: (BECOME NEW-STATEMENT),
       (PRINT (STATEMENT (AND PLEASE_TYPE_IN_THE
                              OBJECT:))),
       (READ OBJECT:)]
```

This declares that an I/O unit is about to begin, that is, a new statement which will end at the end of the METHOD unless otherwise declared. Then it will print a statement of the form "Please type in the *property of the datum*". Finally, it will read the reply from the user. Reading the reply also implies doing a type check to make sure that it really looks like one. For STATE there several METHODs, which all start with a NEW-STATEMENT declaration, followed by the printing of an appropriate statement. The differences are in the contents of the statement. For identifiers it prints "For *identifier*

equal to *identifier value*". For contents it prints "The *property* is *property value*". For the value it prints "The *datum* is *datum value*". The problem for all of these is how to produce the statement.

STATEMENT is an ACTIVE concept, meaning that the evaluator will dispatch to its handler when the concept (PRINT (STATEMENT ...)) is evaluated. The handler is the part of the I/O model that composes the statements. It actually needs to look at the other nodes before and after it, so by forcing the evaluation to be suspended it waits for evaluation until after the following node in the METHOD has been evaluated. It first builds a list of what should be output. If the previous node declares a new statement, the first thing is a carriage return. After that STATEMENT goes through its arguments with a space between each one. When STATEMENT comes across an argument for a datum property name, it must decide what will be printed out. The user knows what the program is to do from the specification, so there is no reason to print out information that is already in the specification. Thus, STATEMENT strips off specializations from the concept that are part of the specification. For example, (ACCOUNT A-DEPOSIT*1) would be printed as ACCOUNT if A-DEPOSIT*1 is a part of the specification. Some concepts also have prefered names to be used for printing -- the prefered name for ETIME is TIME_OF_OCCURRENCE. Finally, if there is a READ as the next node, there must be a space at the end.

Given the list of the desired output, the next step is to combine it into units that can be handled by PRINT and PRIN1. PRIN1 just prints its argument, while PRINT prints a carriage return, its argument, and a space. Two situations that will be encountered are:

1. (<cr> PLEASE_TYPE_IN_THE <sp> ACCOUNT <sp>)
2. (<cr> THE <sp> BALANCE <sp> IS <sp>
((BALANCE ACCOUNT*1)(RESULT ...)))

The routine tries first to fit in PRINT and then uses PRIN1 if that is not possible. It is also considered fair to add a space at the end if that makes PRINT usable. Thus, in the first case the three middle arguments can be combined into a single symbol and PRINT used. The second case requires two print calls, one to PRINT "THE_BALANCE_IS_" and the other to PRIN1 the argument from the desired node.

It is still necessary to do the READ for (ASK ...). Reading is handled by the Lisp READ, but it is also necessary to test the result to see if it passes the basic requirements for such a concept. This is basically a type check. If the result is supposed to be a number then NUMBERP will be applied to it. If it is the name of something, SYMBOLP will test that it is a Lisp symbol. A search is done on the description of the value of the concept for a test which can be used to check the validity. If the test fails something must be done to invalidate the result of the read. The simplest recourse is to return from the program with an error message.

This completes the planning activity of the I/O model, but the I/O model has a stake in the order of the final program. First, the I/O statements must stay in the same order as the order in the tree or the careful work of refinement will be defeated. This is handled by considering any READ, PRINT, or PRIN1 CODE-METHOD to be constrained with respect to the other I/O operation. Secondly, if a program is going to do an error exit, it should do it before doing any more I/O operations. This is handled by giving error returns a high priority in the coder's sorter.

The I/O model presented here is rather simple in comparison to some of the I/O schemes used in large programs, where buffering, files, correction, and interrupt routines add more complications. The main point that should be noted about any I/O model is how the view of the program can differ from the argument view. To try to limit

the I/O capabilities so the routines fit easily within an argument model means forgoing many of the objectives of I/O.

5.1 Interfaces with Other Models

The interface between the I/O model and the other models occurs exclusively through shared terminology. The ASK and STATE METHODs provide the interface to the argument model. These METHODs refine the I/O operations using the same control concepts as any other METHOD, providing the proper instructions on how the order and branching should be treated in the design tree. The INPUT and OUTPUT characteristics provide the interface to the data model. These tell the data model what the overall effect of the operations will be in terms of the data. The PROPERTY and VALUE concepts are the interface with the domain model. To the I/O model PROPERTY means the concept can be asked for by the property name. To the domain model, it means the concept is a significant enough part of the corresponding event to be included in the data abstraction. VALUE is a property name to the domain model, but to the I/O model it means a particular format is appropriate for stating. The interface with the target language model is the characteristic I/O on the primitive operations that do I/O. To the target language model this means a particular kind of side effect such that all of the operations marked I/O have to maintain their relative ordering in the final program. The target language model handles all of the primitive operations in the same way as it does for the argument model, but this is really an interface between the argument model and the target language model.

Section 6 The Target Language Model

The target language model knows about the relationship between the program structure and the target language constructs that will implement the structures. For the programwriter the target language is Lisp[5]. Thus, this is the Lisp model of the program. Many of the problems, situations, and strategies that will occur at the low level will have aspects peculiar to Lisp, but if some other language had been the target language, the same would have been true. In fact the reason a target language model is important is to take advantage of the set of features provided by the target language.

The view of the program taken by the target language model is of a set of constructs that must be organized into a Lisp program (i.e., into a DEFUN). The model has a collection of constructs it knows and understands. The program design tree has a set of requirements in the form of low level ACTIVITYs. The target language model fits its constructs to the requirements.

The programwriter makes use of the model in several ways. The Lisp level implementations of data structure is the first place. After the refinement process has gotten as much out of the data base constructs as it can, Lisp structures must be chosen that will represent the data base parts. This is the first step toward setting up a program. The next entry into the Lisp model is to choose the forms that will operate on the Lisp structures. These are handled through the (SELECT (METHOD ...)) goals. The METHODs at this level are of a special kind, called CODE-METHODs. The STEPS of a CODE-METHOD are patterns for the Lisp code to carry out the function. The final entry into the Lisp model is the coding routine. It is the last step of the programwriter after

[5]To be specific, MACLISP [Moon 1974]

the analysis and planning loop. It takes the design that has been sketched out and turns it into a Lisp program. To do this it must arrange for variables, select orderings where the constraints do not force an ordering, arrange the conditionals to simplify the predicates, make local decisions to take advantage of the available Lisp constructs, and fill out the overall lexical structure of the program.

6.1 Implementing at the Lisp Level

In the end, all of the abstract data base structures must be turned into Lisp structures. This is the final stage of data refinement. This is the level of implementation where the answers will be provided to any questions about the behavior of the structures in the programs. The structures at this level are Lisp structures, but they are specialized by their semantic use. For example, there are different kinds on list for use as association lists or as structures with fixed meanings for the positions. These different kinds of structures have different properties that are important to the METHODs, even though they have the same primitive Lisp structure. For example, a three entry list structure always has three elements if it exists at all. Thus, using the Lisp subroutine CADR (get the second element) on it will never fail, but on an association list it might fail.

The data base parts to be implemented include the DATA-BASE, the DATA-SECTs, the KEY-DATA-SECTs, the RECORDs, and the ITEMs. Each of these present an interesting set of problems for the refinement. As a group they present interactions that were not apparent at a higher level. For example, each Lisp structure except those at the top level serves two purposes. It is a structure containing its parts in some way; and it is a specified part of the structure above in the data base hierarchy.

The distinguishing feature of a DATA-BASE is that it is a global entity. That is, it exists between executions of programs. To implement a DATA-BASE in Lisp, it is necessary to have something that will exist external to the programs. The obvious choices are free variables, property lists, and external files. Only the first two will be considered because files are full of operating system dependent details which are not particularly illuminating. The variables and property lists offer some interesting contrasts. As mentioned before, variables cause a Lisp error if the value is retrieved before it has been set while retrieving a property from a property list that has not been assigned just produces NIL. This means that a variable must be tested before it is used if there is any chance that it may not have been set. On the other hand the NIL returned from the property list could be a legitimate value or an indication that there is no value. (There is actually a way to tell the difference, but that will be ignored.) Thus, property lists may require initialization. The difference between variables and property lists that a human programmer might use to make the choice is the likelihood for conflict. It is very hard to judge such a distinction, so for the purposes of the programwriter the choice is made manually.

Implementation of data bases as either variables or property list properties is analogous to the implementation of sets above but simpler. The implementation of a property list property (a PL-PROPERTY) takes the DATA-BASE, its NAME, and its DATA and must produce a representation for a PL-PROPERTY specifying the PL-NAME, PL-INDICATOR, and VALUE and their correspondence to the DATA-BASE parts. There is one more part in the PL-PROPERTY than in the DATA-BASE, so that is given a standard name. The DATA-BASE itself corresponds to the whole PL-PROPERTY. That is, the notion of its existence. The DATA corresponds to the VALUE of the property. This does not limit the range of structures the DATA could take on as long as they will fit as a value

on a property list. This correspondence specifies the access path to the DATA and by implication the requirements for ASSERTing it.

Using a global variable presents almost the same problems for implementation except that there is only one identifier. The DATA-BASE becomes the global variable in the sense that the variable is an entity for holding onto a Lisp object. The DATA becomes the VALUE of the variable and the NAME of the DATA-BASE can be used for the symbol of the variable.

The rest of the implementations are different uses of lists, since these are the primary structured construct in Lisp. The basic choices are listed below. The choices indicated with an asterisk are the choices made for the basic program. Thus, any of the others must come about through the use of an IDEA.

DATA-BASE	PL-PROPERTY *
	FREE-VARIABLE
DATA-SECT	ASSOCIATION-LIST *
	HEADED-ASSOCIATION-LIST
	MEMBER-LIST
	HEADED-MEMBER-LIST
KEY-DATA-SECT	DOTTED-PAIR *
	MEMBER-PAIR
RECORD	LIST *
	HEADED-LIST
ITEM	N-LIST *

The ASSOCIATION-LIST is a list of dotted pairs of the form that the Lisp subroutine ASSOC can use for retrieval. The MEMBER-LIST is an even length list with the alternate elements paired, suitable for use by the subroutine MEMBER. The MEMBER-PAIRs are the paired elements of a MEMBER-LIST. The structures prefixed by HEADED are versions of the other structures with a dummy element at the beginning. The N-LIST is a list with a specific number of elements, which is a property of the structure. (This only applies if the ITEM has more than one content, otherwise the cell provided by the

container is used for the content.) There are many other ways that lists, not to mention arrays, could be used and normally are by Lisp programers, but these are sufficient to show the possible variations in structure and problems.

Implementing a DATA-SECT and the KEY-DATA-SECTs that it contains is done by a single schema. This is specifically done to take advantage of the interactions between these two structures and such operations in Lisp as ASSOC on ASSOCIATION-LISTs and MEMBER on MEMBER-LISTs. A list whose elements are dotted pairs of identifier and value can be handled very nicely by ASSOC, but it points out an interesting wrinkle in design by refinement. At this point in the design, the DATA-SECT and the KEY-DATA-SECT are separate parts in the DATA-BASE structure, but to take advantage of these operations they must be considered together. This kind of situation occurs often in programming and leads occasionally to the reconsideration of decisions that were made too locally. In this case the problem can be avoided by considering the implementations in a fixed order, and taking advantage of that order in the orientation of the SCHEMA. Thus, the goals for the DATA-BASE can be done in the order of the hierarchy, with the SCHEMA for DATA-SECT recognizing that it also should handle the KEY-DATA-SECT that is a part of the DATA-SECT. In other situations the order might not be so clear or one of the goals might be held up for other reasons. To handle the general case it is necessary to be able to back up.

At the Lisp level, the all of the methods used to refine the final offerings from above are CODE-METHODs, providing the code to accomplish the tasks. Because the problems have been fairly well broken down at this point, most of these CODE-METHODs are single Lisp subroutines. For example, assertion on a LIST is done with CONS, creating a new LIST. Assertion of the second part of a DOTTED-PAIR is done with

RPLACD, changing the DOTTED-PAIR. Iteration is accomplished with DO, incorporating the BODY of the ITERATE into the body of the DO. At this level most of the refinement is relatively straight-forward. Besides the Lisp forms, the CODE-METHODs also provide characteristics for the results that help to clear up any pending decisions from above.

Once all of the nodes (except for nodes suspended for the coding phase) have been turned into EVENT-CODE nodes, the analysis and planning phase is complete.

6.2 Coding

The last phase of the programwriter is the production of code from the Owl structure built up by the analysis and planning. The terminal nodes of the structure are EVENT-CODE nodes that give the Lisp forms needed for the code. The coder goes over the structure taking these CODE-METHODs, the constraints on their order, the requirements for their arguments, and uses them to produce the code. When the coding takes place, the steps of the programs are not necessarily completely constrained. For example, the arguments to a DIFFERENCE node (the activity for doing a subtraction) can be computed in either order without affecting the semantics of the program. Therefore, they are connected by AND rather than THEN. Because of the argument handling in Lisp it is easier to do the minuend first, thus not requiring an auxiliary variable.

A step of a CODE-METHOD is either a LISP concept, a LISP-VALUE, or a BECOME. The LISP and LISP-VALUE produce pieces of code for the program, while the BECOMEs provide information for the design. The step in the CODE-METHOD for retrieving a value from a property list is

```
(LISP GET (PL-NAME PL-PROPERTY:)
          (PL-INDICATOR PL-PROPERTY:)).
```

This is a LISP concept representing the Lisp list that applies GET to the name and indicator of the property list. The PL-PROPERTY: is a variable that is the FROM of the CODE-METHOD. It is bound to the PL-PROPERTY implemented to correspond to the DATA-BASE. From the implementation the evaluator can determine the PL-NAME and PL-INDICATOR. Thus, when the concept is evaluated to produce a piece of Lisp structure for the program it will produce (GET 'DB-DEPOSIT 'DATA-BASE). To produce this the coder needs to know not only about the values of the concepts, but also that GET is a *subr* (a subroutine that evaluates its arguments) and will need these symbol arguments quoted.

A LISP concept can have another kind of argument, the result of some other step such as (AMOUNT (A-DEPOSIT*1 (RESULT (ASK ...)))). This presents some interesting complications. Obviously at the Lisp level the programwriter wants the result of the actual CODE-METHOD node that does the Lisp READ, which is the terminal node several levels below (ASK ...). To get the appropriate piece of Lisp, the coder has to descend the tree picking the step at each level that claims it will return the thing of interest.

Several things must be considered in the process of producing the Lisp, besides the ordering of ANDs and the problems of quoting. Several pieces of Lisp may want the same result, so a variable or some other arrangement will have to be made to provide it in several places. Also, the variable assignments left over from above must be handled.

The constraints on the order of program statements come in several flavors. One is as a result of I/O. To maintain the desired I/O behavior of the program, the statements that do I/O must be executed in the same order in the program as they

appear in the tree. This constraint is reflected by BEFORE properties generated by the coder. Another constraint is that imposed by the arguments. That is, any statement that produces the argument for another statement must come before the second statement. Finally, the THENs used throughout the tree impose a constraint. This is actually not a very strong constraint. Even though ASKing must come before ASSERTing in the ACCEPT procedure in the introductory example, all of parts of ASSERTing that do not need the result of ASKing can come earlier. For example, retrieving the RECORD the datum will go in can take place before or during the steps of ASK. The times when this constraint is needed (besides the I/O nodes) are cases of a change being made to a particular Lisp structure. If a RPLACD changes a DOTTED-PAIR, the retrieval of the SECOND-PART must remain before the RPLACD. This problem is recognized by checking the

(BECOME (SECOND-PART (DOTTED-PAIR: THE)) <- CHANGE)

property declared by the CODE-METHOD.

There are also some special considerations involved with CONDs, assignments, error exits, and DO loops. For CONDs predicate test may be reordered to simplify the construction. The coding routine has a few simple modifications it can use to switch the clauses to eliminate NULL tests, or simplify other logical constructs. The assignments are all done with SETQ except for one case. If the variable is only used in the body of an iteration, the initial assignment is moved to the initialization part of the DO. The error exits should happen as soon as possible. In particular, nothing should be input from the user after it is possible to make an error exit from the program. This is handled by giving high priority to error exits in the sorting of the final program. For the DO loop, a check is made for nodes that do not involve the element concept or otherwise constrained to be there (i.e., they do I/O) and they are moved out of the DO.

Given all of these constraints, the statements are put into a partial order sorter to find out what temporary (e.g., PROG) variables are needed. The sorter tries to put statements and their arguments together so PROG variables are not needed, but that is not always possible. In cases where two statements use the same argument, a prog variable is needed unless one of the statements returns its argument. Occasionally, order constraints dictate that a statement come between an argument and its use. Occasionally, the second argument to a statement must be generated before the first. In any of these cases the coder generates a PROG variable and installs the SETQ to handle the situation. It must also set a flag specifying that the whole program will be inside the PROG. This flag may already have been set if the program needed an error return at some point.

6.3 Interfaces with Other Models

The interface between the target language model and the others is one way, since it is the last model to do its work. The target language model of the program produces implementations of the data base structures, made possible by an understanding of the target language implications of the data base constructs. That is, it understands the concepts, but only in terms of how they will constrain the choices of Lisp constructs. The model also provides Lisp operation constructs (the CODE-METHODs) to carry out the ACTIVITYs. This is really an interface to the design model, rather than one particular model. Finally, the model combines the whole works into the Lisp form of a program. To be able to do this requires an interface with the argument model. The target language model needs an understanding of the control primitives in terms of what they mean as constraints on the final program.

Chapter IV
Scenario
for a Savings Account Program

To show how the various mechanisms and structures fit together to make the programwriter operate, this chapter will present scenarios of the programwriter designing sets of programs. There are four scenarios of which the last three are variations of the first. The first scenario does a very basic job of designing a simple savings account system. The second scenario is a modification of the first with ideas considered to improve the efficiency of the code. The third scenario is a modification of the specification for the first to show the effects of alternative situations. The final scenario adds efficiency ideas to the third scenario. The scenarios are fairly detailed and intended to show how the programwriter goes about the job of designing programs, the ways the models are used, the kinds of information it uses, and the kinds of decisions it needs to make.

0.1 A Preview of the First Scenario

Before delving into the details of the scenario, it is appropriate to consider a sketch of what will take place without going into the reasons for the considerations and decisions. The following section will give a tour of the first scenario. Of necessity much has been simplified, but the description should be sufficient to provide perspective during the detailed sections to follow.

The request to the programwriter is for three programs: one to accept deposits from the user, one to accept withdrawals, and one to state the balance of an

account. The programwriter will go through its main loop of analyzing the situation, choosing a goal to pursue, and dispatching to a module to handle the goal. As a result, the first thing that happens is to set up the initial nodes for the three programs. (We will not talk about the accept withdrawal program, because it is similar to the accept deposit program.)

(ACCEPT A-DEPOSIT*1) (STATE (BALANCE ACCOUNT*1))

Figure 20. Initial nodes

The first discovery is made by the argument model, while analyzing the STATE program. There is no node that will provide the balance to be stated. To do that, a node is added to the STATE program that will produce the balance before attempting to state it. The next step is to determine what it means to "produce the balance". The domain model provides a definition of the balance that fits the circumstances declared in the specification. The definition says the balance is the difference between the initial balance plus the sum of the deposits and the sum of the withdrawals. The argument model checks this definition with an eye toward computing the balance in this way. To do so would also mean that the initial balance, the set of deposits and the set of withdrawals would have to be available for the computation.

At this point the data model takes over to determine how the data is going to be used by the programs and what storage requirements it will have. The primary data used by the system are the deposits and withdrawals. They will come from the ACCEPT programs and will be used by the STATE programs. Because of this, they must be stored in some kind of data base. Thus, it is necessary to look at the deposits and withdrawals in terms of what attributes they will require in a data base. For example,

each deposit (or withdrawal) will be accepted at the time it takes place and no other time. Therefore, once a deposit is received it will never change and no provision need be made for changing one. In the process of checking the data, it is discovered that the STATE program has no way of telling what account the balance is for. To correct that problem, the data model causes a step to be added to produce the account number.

Once the data model has done its work, the situation with the data is much more clearly defined and it is possible to go ahead with the refinement. The ACCEPT node is refined into the sequence: ASK the user for the deposit, then the result becomes the deposit. Because the data model wants the deposit stored, the "become" node is turned into one to ASSERT the deposit. ASKing for the deposit causes the I/O model to take over. The data model determined that the deposit has an identifier consisting of the account and the time it happened (the ETIME). It also determined that the amount of money is the contents. These are the parts of the deposit that ASK will have to get. Actually, it will not have to ask for the ETIME because it knows that the execution of the program will take place at the same time and can get that time with a system call. The I/O model knows that it is more convenient to the user to ask for the identifier first. The way the I/O model ASKs for a specific property is to print a request to the user and read the reply, which includes checking to see that the result fits the basic requirements for that property. Once the I/O model has determined what to print, the target language model can provide actual code for all of the operations.

At this point, the ACCEPT program has the following form:

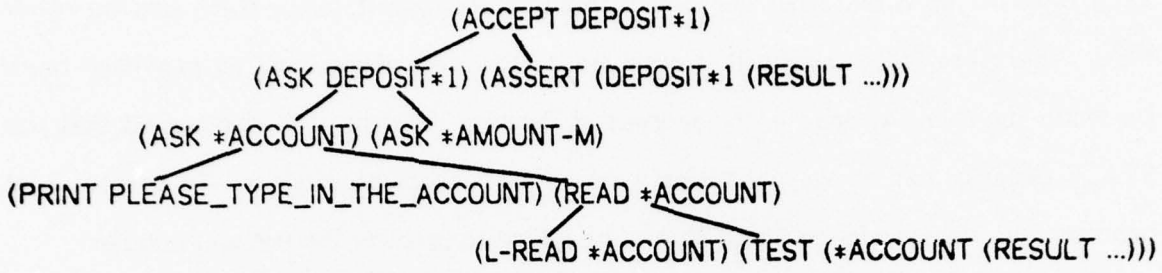

Figure 21. Sketch of ACCEPT

The (ASK *AMOUNT-M) portion is also refined in the same way as the (ASK *ACCOUNT) portion.

Next, the STATE program will be refined. From the information provided by the data model it is concluded that computing the definition is the appropriate way of producing the balance. The argument model goes ahead and uses this to refine the producing of the balance. There are still some requirements for arguments to the computation which must be handled before the computation can be further refined. The information needed to decide how to produce these arguments is available because of the previous work of the data model. The account number can be produced by asking the user, since it is an identifier. The deposits and withdrawals needed for the computation will have to be retrieved from those stored by the ACCEPT programs. The initial balance will not have to be retrieved at all because it is always zero. With this information, refinements of the computation can be chosen, taking advantage of the facts known about the arguments.

The second part of the program is to do the STATEing. This is handled by the I/O model, taking into account what the user already knows to determine what should be stated about the balance. The result is that it will print out "THE_BALANCE_IS_" followed by the value computed in the first part.

At this point the STATE program looks as follows:

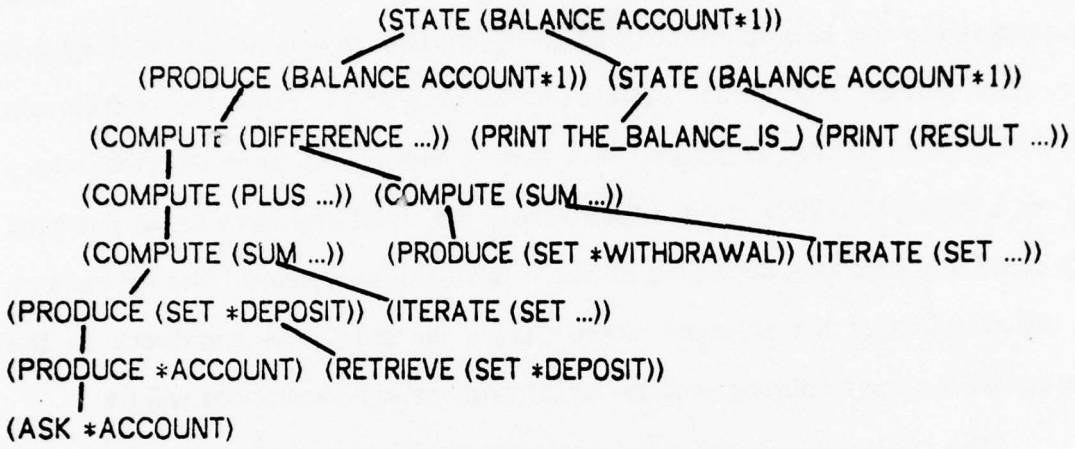

Figure 22. Sketch of STATE

It is now time for the data model to take over again and refine the storage medium to be used for the retrieving and asserting of the data. The data model has a general plan for designing data bases, which can take into account the normal kinds of requirements encountered. It decides on a two level data base, reflecting the way the identifiers are used. The deposits of a particular account are implemented as indistinguishable items in a single record. These records, paired with the account they are for, make up the data. The different parts of this structure have names to assist in the refining of the operations. The data is called a DATA-SECT and each pairing of an account and a record is called a KEY-DATA-SECT.

With the new data base structure provided by the data model, it is now possible to refine the assertions and retrievals. To assert a new deposit into such a data base, it is necessary to find the record in which it belongs and then insert it into the record. Finding the record involves getting the data from the data base, finding the

right KEY-DATA-SECT and accessing the record. Because the data base is built up as deposits are put into it, it is possible that the KEY-DATA-SECT (and therefore the record) for the deposit do not exist yet. The argument model must make provision for such problems and requests information from the data model on whether or not they will happen. The solution is to include a branch in the program to create the appropriate pieces of structure in case of failure. Also, once a deposit has been inserted into a record (or a KEY-DATA-SECT into the DATA-SECT) the result may have to be put back into its containing structure, depending on how it was actually inserted. The refinement, under the direction of the argument model, designs the data base operations to the degree possible without knowing what the target language implementations will be.

Once the programs have been refined to this level of detail, the rest of the decisions depend on what will happen in the actual code. The next step is for the target language model to decide on structures for the various pieces of the data base. Taken in the order of retrieval, it is fairly easy to match up the data base constructs with appropriate Lisp constructs. The data base itself can be a property on a property list. (Remember that the program environment is a Lisp environment that exists throughout the execution history, so using the storage facilities inside of Lisp is legitimate.) The DATA-SECT can be an association list with the KEY-DATA-SECTs as the associations. For the records, all that is needed is a list in which to keep the contents of the deposits (i.e., the amounts).

Given the implementation of the data base parts the refinement of the operations follows without difficulty. The Lisp model has provided the knowledge base with a collection of refinements for the desired operations on the various Lisp structures. For example, the retrieval of the association representing a KEY-DATA-

SECT from the association list representing the DATA-SECT will take place using the Lisp subroutine ASSOC. Once the operations have been turned into code, the rest of the programming is up to the Lisp model's coding routine. This takes the collection of operations in the program design tree and the constraints on them and decides on a Lisp program structure for the finished program.

The following figure is a rough representation of the program design tree at the end of the design process with the areas marked where the different models had major influence.

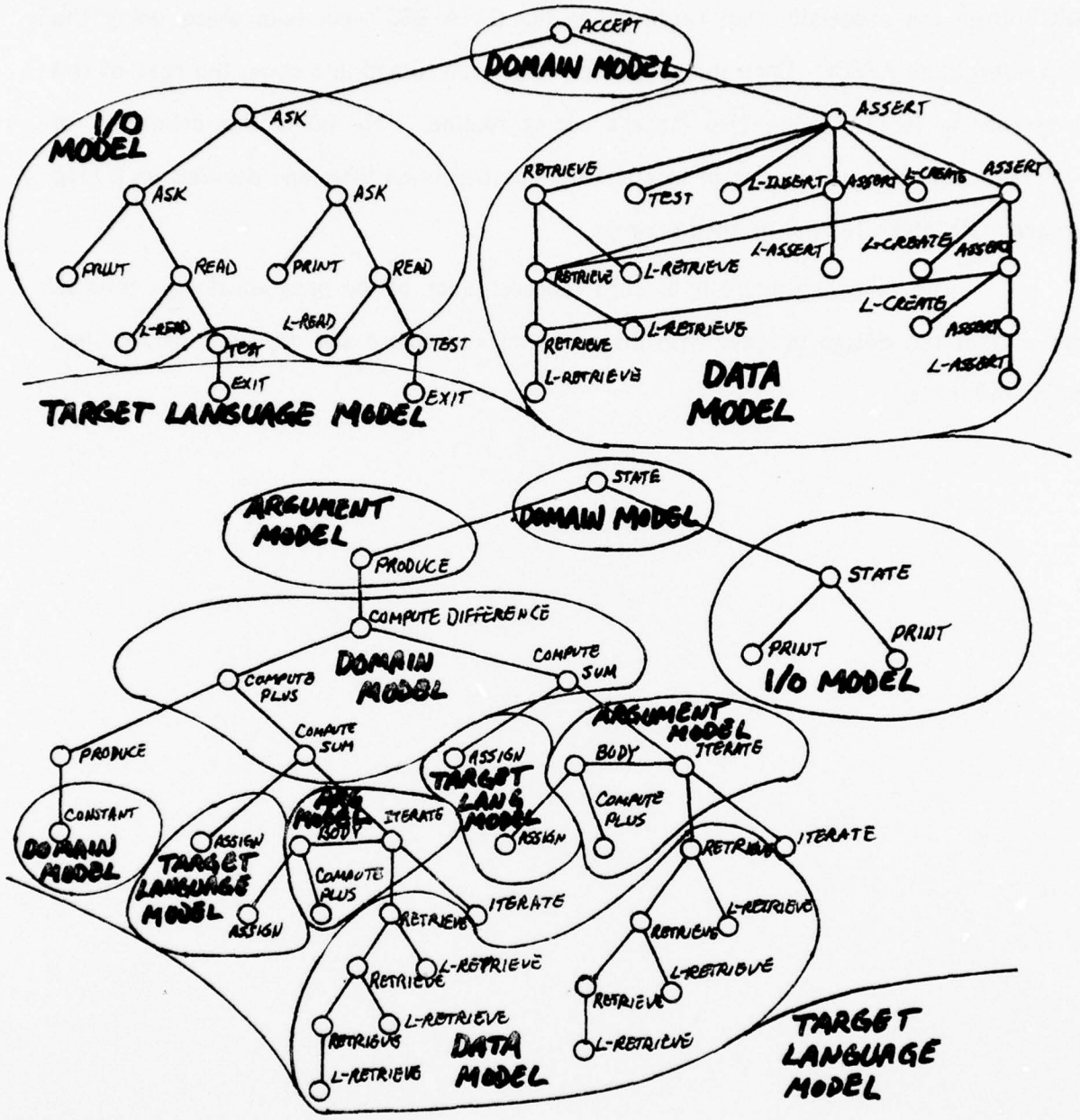

Figure 23. Model influence in program design tree

Section 1 The Basic Scenario

The request to the programwriter is for a set of programs to handle savings accounts. The desired programs will accept a deposit to an account, will accept a withdrawal, and will answer a request for the balance of some account. The specification appears as follows:

```
[(WRITE [(SET (ACTIVITY "SAVINGS-ACCOUNT")) <-
        (AND (STATE (BALANCE ACCOUNT*1))
             (ACCEPT A-DEPOSIT*1)
             (ACCEPT A-WITHDRAWAL*1))])]
```

The program requests are represented as the elements of an explicit set, but the programwriter can change the elements if it needs to include more than the three programs to accomplish the goal. Each of the program requests could have constraints on it indicating restrictions on the times of execution, or requirements for the specializers, but this set of specifications do not. Since there are no constraints given, the possible times these programs may be executed are assumed to be *unconstrained*. That is, any order is possible and there are no requirements for the deposits, withdrawals, and accounts beyond what is implied by the concepts themselves. The A-DEPOSIT and A-WITHDRAWAL are really labels for the concepts "the act of depositing money" and "the act of withdrawing money", respectively. A-DEPOSIT can be represented as:

```
[A-DEPOSIT = (ACT (DEPOSIT MONEY))
           IDENTIFIER: <- (AND [ETIME: PROPERTY]
                               [ACCOUNT: PROPERTY])
           [AMOUNT-M: PROPERTY]
           [BANK-BRANCH: PROPERTY]]
```

That is, the identifier of a deposit is the combination of time of execution of the deposit and the account (two deposits are the same only if they have the same time of execution and account), and a deposit has an amount of money and a bank branch

associated with it. Each of these parts of an A-DEPOSIT are PROPERTYs. The "*1" on each concept in the specification indicates it is a dummy standing for a specific instance of the concept. Any particular execution of the (STATE (BALANCE ACCOUNT*1)) program (or either of the others) will deal with a single specific account.

The balance concept has the following properties in the knowledge base:

```
[(BALANCE ACCOUNT)
 DATUM
 IDENTIFIER: <- [ACCOUNT: PROPERTY]
 CONTENT: <- [VALUE: PROPERTY]
 [DEFINITION:
  OBJECT:: <- (BALANCE ACCOUNT::)
  RESULT:: <-
   (DIFFERENCE
     (PLUS ((BALANCE (ACCOUNT:: THE)) INITIAL)
          (SUM (SET (AMOUNT-M [A-DEPOSIT:: PATTERN
                               ACCOUNT::: <- (ACCOUNT:: THE)])))))
     (SUM (SET (AMOUNT-M [A-WITHDRAWAL:: PATTERN
                          ACCOUNT::: <- (ACCOUNT:: THE)])))))]
 ((BALANCE ACCOUNT) INITIAL) <- 0]
```

The concept stands for the balance of accounts, in general. The identifier for a BALANCE of an ACCOUNT is the particular ACCOUNT and its content is the value of the balance. There is also a definition that defines the relation of the balance concept to the deposit and withdrawal concepts. The initial balance of an account is known to have the value zero. BALANCE also is a kind of AMOUNT, so its value will always be a number.

Given this information the programwriter is ready to start. The initial analysis step sets up a program node for each of the desired programs and binds these nodes to the corresponding INTENTs. The INTENT for (STATE (BALANCE ACCOUNT*1)) provides the following:

```
ARGUMENT: (BALANCE ACCOUNT*1)
OUTPUT: (BALANCE ACCOUNT*1)
```

That is, (BALANCE ACCOUNT*1) is an argument of the node and will be output by it on

the console. Because the argument is a DATUM, the node will be marked as a USE of the DATUM and the data model will want to have a look at it. For the data model, the analysis module sets up a goal to CHARACTERIZE the balance. The concept (BALANCE ACCOUNT*1) is a dummy standing for the balance of *some* account while the data module actually needs to consider all such balances. Therefore, the goal is set up in terms of the root concept (BALANCE ACCOUNT) with the actual form encountered noted on the reference list. The analysis also needs to set up a goal to (SELECT (METHOD (STATE (BALANCE ACCOUNT*1)))), but the goal to CHARACTERIZE will take precedence. That is, the CHARACTERIZE goal will be set up as:

```
[(CHARACTERIZE (BALANCE ACCOUNT))
 FOR: <- (BALANCE ACCOUNT*1)
 (BEFORE (SELECT (METHOD (STATE (BALANCE ACCOUNT*1)))))]
```

Because the INTENT claims (BALANCE ACCOUNT*1) is an ARGUMENT, it has to come *from somewhere*. That is, the argument model requires that every argument to a node come from some other node in the program. However this is a top node, so there is no prior node of the program that could return the value. To handle this problem the analysis decides a node must be added to produce the balance. The goal that is set up is to make a PREMETHOD before selecting a METHOD.

```
[(MAKE (PREMETHOD (STATE (BALANCE ACCOUNT*1))))
 TO: <- [(PRODUCE (BALANCE ACCOUNT*1))
        (BEFORE (STATE (BALANCE ACCOUNT*1)))]
 (BEFORE (SELECT (METHOD (STATE (BALANCE ACCOUNT*1)))))]
```

Similarly, analyzing the two nodes for accepting deposits and accepting withdrawals causes them to be bound to the appropriate INTENTs and examined. For example, the INTENT for (ACCEPT A-DEPOSIT*1) states:

```
INPUT:  A-DEPOSIT*1
SOURCE:  A-DEPOSIT*1
KNOW:  [(ETIME A-DEPOSIT*1) <-
         (ETIME (ACT (ACCEPT A-DEPOSIT*1)))]
```

Thus, the node will input the deposit from the user. The SOURCE statement means that the node can act as the source for any deposits that would match the A-DEPOSIT*1 dummy. That is, if A-DEPOSITs are needed elsewhere, this node could be the place that produces them. SOURCE is a term understood by the modules of the data model used to identify the point in the program where the datum is created. The use of A-DEPOSIT*1 as a pattern indicating the possible deposits for which this could be a source is a reflection of the difference in viewpoint of the data model and the argument model. From the point of view of the argument model, this node is the source of the particular deposit the program is accepting, which is what the statement says. Any source is of interest to the data model, which may have requirements for the node to accomplish. Again, the data model is actually interested in the root concept, so the analysis module creates a goal to (CHARACTERIZE A-DEPOSIT). Again, it is also necessary to (SELECT (METHOD (ACCEPT A-DEPOSIT*1))), but the CHARACTERIZE goal will take precedence. That way the data model can supply enough information to make possible the selection of the appropriate METHOD. There is also a KNOW property indicating that the time of execution of the A-DEPOSIT*1 will be the time of execution of the ACCEPT procedure. This is something known by the domain model about "accepting" that distinguishes it from "correcting". It is the convention of the programming domain that the user runs the program at the time the depositor makes his deposit. This information will be useful not only in designing the programs for accepting, but also for determining what properties the data base representation for A-DEPOSIT*1 will need.

With the INTENTs for the three nodes expanded and analyzed, the programwriter is finished with the first analysis step and proceeds to a planning step. Planning consists of selecting a goal, finding a procedure to handle the goal, and executing it. In this case, the goals that are open to pursue are the ones to

CHARACTERIZE data and the one to make the PREMETHOD. The rest are not open because there are other goals constrained to come before them. Making the PREMETHOD is handled first because the goal has higher priority. The planning module dispatches through the knowledge base to the module for (MAKE PREMETHOD), which is part of the argument model. It produces the following:

```
[(PREMETHOD (STATE (BALANCE ACCOUNT*1)))
 STEPS: *PRODUCE=(PRODUCE (BALANCE ACCOUNT*1)),
        (STATE ((BALANCE ACCOUNT*1)(RESULT *PRODUCE)))]
```

The PREMETHOD is then bound to the top level node as its METHOD. To end the planning step, the goal (MAKE (PREMETHOD ...)) is marked COMPLETE. This action has actually completed the SELECT METHOD goal also, but when that goal is chosen the existing PREMETHOD will be discovered and it will be marked COMPLETE.

Each step of analysis after the first is only concerned with those parts of the program structure that have changed. This time it is the PREMETHOD. Each of the steps are handled in the same way as the top node. The INTENT for PRODUCE wants to know if there is a DEFINITION for (BALANCE ACCOUNT*1). Finding the one on (BALANCE ACCOUNT), it makes a goal to EXPAND the DEFINITION. It also wants to (CHARACTERIZE (BALANCE ACCOUNT)) for (BALANCE ACCOUNT*1), making the same goal as was produced for the STATE call. Both the EXPAND goal and the CHARACTERIZE goal must happen before the corresponding SELECT METHOD goal. The only other fact provided by the INTENT is that the node will return (BALANCE ACCOUNT*1). The analysis of the new (STATE ...) is the same as before, except that it is now satisfied with the argument.

The next planning step selects the goal to EXPAND the DEFINITION. Evaluating the DEFINITION of BALANCE in the present situation, turns it into the following:

```
[(DIFFERENCE (PLUS ((BALANCE ACCOUNT*1) INITIAL)
                   (SUM (SET (AMOUNT-M [A-DEPOSIT*2 PATTERN
                                        ACCOUNT:: <- ACCOUNT*1])))
                   (SUM (SET (AMOUNT-M [A-WITHDRAWAL*2 PATTERN
                                        ACCOUNT:: <- ACCOUNT*1])))]
```

This definition provides the connection between the balance, the deposits, and the withdrawals needed to analyze the data dependencies in the set of programs. To make use of it will require analysis. It is added as a property to (PRODUCE (BALANCE ACCOUNT*1)), and the planning step ends by marking as COMPLETE the goal to EXPAND the DEFINITION.

The analysis of the definition ultimately means each part of the expression must be matched to the appropriate INTENT to know what it means for the program. This is accomplished by repeatedly applying INTENTs to the parts of the expression still unexplained. Since the definition is an arithmetic expression, the way to produce it is to compute it. Because of this, the INTENTs are written in terms of computing the expression. The analysis module therefore forms the concept (COMPUTE (DIFFERENCE ...)) and analyzes that. The INTENT for (COMPUTE DIFFERENCE) produces:

```
ARGUMENT: (AND (PLUS ((BALANCE ACCOUNT*1) INITIAL)
                     (SUM (SET (AMOUNT-M [A-DEPOSIT*2 PATTERN
                                          ACCOUNT:: <- ACCOUNT*1]))))
                     (SUM (SET (AMOUNT-M [A-WITHDRAWAL*2 PATTERN
                                          ACCOUNT:: <- ACCOUNT*1]))))
```

These two ARGUMENTs are also arithmetic expressions and must be analyzed by forming the (COMPUTE (PLUS ...)) and (COMPUTE (SUM ...)) expressions. The INTENT for (COMPUTE PLUS) claims ((BALANCE ACCOUNT*1) INITIAL) and (SUM (SET (AMOUNT-M A-DEPOSIT*2))) are ARGUMENTs. The INTENT for (COMPUTE (SUM SET)) applied to (COMPUTE (SUM (SET (AMOUNT-M A-WITHDRAWAL*2)))) claims (SET (AMOUNT-M A-WITHDRAWAL*2)) is an argument, and then applied to (COMPUTE (SUM (SET (AMOUNT-M A-DEPOSIT*2)))), that (SET (AMOUNT-M A-DEPOSIT*2)) is an argument.

Analyzing ((BALANCE ACCOUNT*1) INITIAL) is similar to the analysis of (BALANCE ACCOUNT*1) as an argument. It is marked as a USE of ((BALANCE ACCOUNT*1) INITIAL) and must be examined by the data model. Thus, a goal is set up in terms of the root concept, (BALANCE ACCOUNT), to characterize it. This is actually the same goal as before, which now has:

```
[(CHARACTERIZE (BALANCE ACCOUNT))
 FOR: <- (AND (BALANCE ACCOUNT*1)
              ((BALANCE ACCOUNT*1) INITIAL))
 ...]
```

It also must come from somewhere, causing a goal to make a PREMETHOD to (PRODUCE ((BALANCE ACCOUNT*1) INITIAL)). This all takes place as before, except that there is no DEFINITION for ((BALANCE ACCOUNT*1) INITIAL). Thus, further details must wait until the characterization of data has taken place.

Analyzing (SET (AMOUNT-M A-DEPOSIT*2)) is also similar (as is (SET (AMOUNT-M A-WITHDRAWAL*2))). It must be characterized in terms of the root concept A-DEPOSIT. It must also be produced. The goal to set up a PREMETHOD to produce it results in (PRODUCE (SET (AMOUNT-M A-DEPOSIT*2))). There is no definition, so no expansion needs to be done. Thus, everything has been done to analyze these concepts that can be done until the characterization provides more information. Each of the arithmetic expressions has a property explaining it in terms of the computation required and all of the data is waiting for characterization.

At this point all of the open goals are to CHARACTERIZE data. This is an entrance into the modules of the data model. The characterization of data is one of the major functions of the data model, establishing the basic creation and usage pattern of the data in this set of programs. Since there is no prefered order among these three goals, consider:

```
[(CHARACTERIZE A-DEPOSIT)
 FOR: <- (AND A-DEPOSIT*1
              (SET (AMOUNT-M [A-DEPOSIT*2 PATTERN
                              ACCOUNT:: <- ACCOUNT*1])))
 ...]
```

(The handling of (CHARACTERIZE A-WITHDRAWAL) will be analogous.) The purpose of this goal is to find out enough about the datum to be able to implement it. The kinds of things that need to be discovered include: the places it is used, its sources including where it might be changed, its structure, and when it is defined. To characterize a datum at all, the analysis of the existing nodes and of the definitions must be complete, otherwise there might be pertinent facts that would be left out of the characterization. To verify that this is true, a search of the goal list is made for any uncompleted analysis or expansion. If there are any, the CHARACTERIZE goal is suspended pending the completion of the offending goals. With this assurance that complete knowledge exists about A-DEPOSIT, characterization commences. The first step is to determine the SOURCEs of the forms of A-DEPOSIT. The SOURCEs were marked by the INTENTs, and only ACCEPT is a SOURCE of A-DEPOSITs. The SOURCE of a particular A-DEPOSIT is the corresponding execution of the (ACCEPT A-DEPOSIT*1)[1] program.

```
[(SOURCE [A-DEPOSIT#1 *ET=ETIME::
                      *ACCT=ACCOUNT::]) <-
  [(ACT [(ACCEPT [A-DEPOSIT#2
                  ETIME:::: <- *ET
                  ACCOUNT:::: <- *ACCT])])
   ETIME:: <- *ET]]
```

The next step is to examine the USEs of the various forms of A-DEPOSIT. The only USE discovered by analysis is (SET (AMOUNT-M A-DEPOSIT*2)), but this presents some problems. This refers to the set of A-DEPOSITs with (ACCOUNT A-DEPOSIT*2) equal to ACCOUNT*1, but the actual ACCOUNT intended by the user (or to be intended by the user when he runs the program, remember ACCOUNT*1 is a dummy) has yet to be

[1] Remember, the "#" concepts are patterns, while the "*" concepts are dummies standing for a specific concept.

AD-A047 595 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
A PROGRAM WRITER.(U)
NOV 77 W J LONG
MIT/LCS/TR-187
N00014-75-C-0661
NL

UNCLASSIFIED

3 OF 3
ADI
A047595

END
DATE
FILMED
1-78
DDC

determined. This would be all right if a value existed for the AMOUNT-M of A-DEPOSITs in general, but it does not. Without any SOURCE for such a set, the characterization must be suspended until the ACCOUNT*1 has been determined. Since the way to determine it is to produce it in the program, a goal to make a PREMETHOD of (PRODUCE (SET (AMOUNT-M A-DEPOSIT*2))) to (PRODUCE ACCOUNT*1) is set up.

Handling the MAKE PREMETHOD goal results in the following two steps under the (PRODUCE (SET (AMOUNT-M A-DEPOSIT*2))) node:

```
*PROD=(PRODUCE ACCOUNT*1),
(PRODUCE (SET (AMOUNT-M [A-DEPOSIT*3 PATTERN
                         ACCOUNT: <- (ACCOUNT (RESULT *PROD))])))
```

The only difference in the node is that the pattern for the set of A-DEPOSITs is now determined at the appropriate point in the program. The PRODUCE step also results in a goal to (CHARACTERIZE ACCOUNT), which will be handled later.

Now the suspended characterization can be continued. The USEs of A-DEPOSITs now have corresponding SOURCEs. There also may be additional USEs of A-DEPOSITs if the SOURCEs can change existing A-DEPOSITs. That is, to change something is to implicitly USE it. This possibility is checked by looking at the restrictions on the SOURCEs. In this case a property provided by the INTENT of ACCEPT establishes that part of the identifier of the A-DEPOSIT (the ETIME) is the ETIME of the SOURCE. Since no two program executions have the same ETIME, the A-DEPOSITs can not change and are not USEd by the SOURCE. Thus, the USEs are:

```
[(USE [A-DEPOSIT#3 *ACCT=ACCOUNT::]) <-
 (SET (ACT (STATE (BALANCE *ACCT))))]
```

Because the A-DEPOSITs can not change they are marked (CHANGE NOT). Because there is a USE of the A-DEPOSITs they must be recorded in some way, making them available when they are needed. To assure that it will happen, the SOURCE (ACCEPT A-

DEPOSIT*1) is given a DURING property to (ASSERT A-DEPOSIT*1). This means that when the A-DEPOSIT comes into existence in the ACCEPT program it will be asserted.

There is nothing given among the properties about an initial value for an A-DEPOSIT, so they must be initially undefined. An indicator (DEFINED NOT) is put on A-DEPOSIT to record this fact. The set of identifiers for the set of possible A-DEPOSITs is not a finite, explicit set, so the concept must also be marked (KNOWN NOT). Finally, the parts of the datum that are really needed have not been specified. To determine this it is necessary to see what properties are used in the programs. The identifier is the ETIME and the ACCOUNT, but only the ACCOUNT is used. (ACCOUNT A-DEPOSIT) is marked NEEDED and (ETIME A-DEPOSIT) is marked (NEEDED NOT). The only part of the contents that is used is the AMOUNT-M. Thus, AMOUNT-M is proclaimed to be the CONTENT of A-DEPOSIT.

(SET (AMOUNT-M A-DEPOSIT*3)) must also be characterized. Actually, the set to be characterized is

```
(SET (AMOUNT-M [A-DEPOSIT#4 ACCOUNT: <-
                    [ACCOUNT#1 SPECIFIC]]))
```

which is the generalized version of the desired set. That is, it is a representation of the set stripped of references to specific programs. There is one such set for each ACCOUNT, but the set of ACCOUNTs is not explicitly given. Therefore, the set is marked (KNOWN NOT). All sets, unless specifically initialized, are initially empty, making it DEFINED. And, each execution of (ACCEPT A-DEPOSIT*1) for the given ACCOUNT CHANGEs the set. Finally, the SOURCE is

(SET (ACT (ACCEPT [A-DEPOSIT#5 ACCOUNT: <- ACCOUNT#1]))),

and the USEs are (SET (ACT (STATE (BALANCE ACCOUNT#1)))).

The characterization of (BALANCE ACCOUNT) is handled in the same way. Since the identifier is the ACCOUNT, there are as many balances as accounts. Thus, it is also (KNOWN NOT). Its content is its VALUE, which is initialized to zero, making it DEFINED. Its SOURCE comes from the analysis of the definition and includes all of the A-DEPOSITs and all of the A-WITHDRAWALs. Therefore, it can change and must be marked CHANGE. The ((BALANCE ACCOUNT) INITIAL) must also be characterized since ((BALANCE ACCOUNT*1) INITIAL) must be produced. This time (unlike the problem with A-DEPOSIT*2) the general concept does have a value, making it unnecessary to determine which account is wanted. Since it has a fixed value it is KNOWN, DEFINED, and (CHANGE NOT).

Finally, there is the characterization of ACCOUNT. Since it is one of the identifiers, its SOURCE must be the user, it can (CHANGE NOT) once the user has specified it, and it is KNOWN. There is no initial account value, so it is also (DEFINED NOT).

With these characterizations complete, the goals to SELECT METHODs for the three program nodes are open. All of these are simple requests for which there is a clear choice of METHOD in the knowledge base. These METHODs are bound to the respective nodes and then analyzed. (To make the presentation easier to follow, I am describing the planning steps in an order more intuitive than what might happen with the goal selection process. The goals are actually handled one at a time, followed by analysis, etc.)

Consider the METHOD for (ACCEPT A-DEPOSIT*1).

```
[(METHOD (ACCEPT DATUM))
 OBJECT: <- DATUM:
 STEPS: *ASK=(ASK DATUM:),
        (BECOME (DATUM: THE) <- (RESULT *ASK))]
```

The datum A-DEPOSIT*1 is asked for from the user, and then the result of the ASK procedure BECOMEs the actual datum. Nodes representing the steps are created below the main node for the program. Then, these nodes are analyzed in the same way as the program nodes were. The node for the first step is (ASK A-DEPOSIT*1). This is bound to the INTENT for (ASK (ACT TRANSACTION)), which claims it will INPUT the deposit from the console and RETURN (A-DEPOSIT*1 PROTO).

The METHOD for (ASK A-DEPOSIT*1) ASKs for the NEEDED parts of the value of (IDENTIFIER A-DEPOSIT*1) first, then ASKs for the NEEDED parts of the value of (CONTENT A-DEPOSIT*1). The characterization of A-DEPOSIT determined that the ACCOUNT (one part of the identifier) and the AMOUNT-M (the only CONTENT) are NEEDED. Thus, each step of the METHOD results in a new node for the program. The first step is to (ASK (ACCOUNT A-DEPOSIT*1)) and the second step is to (ASK (AMOUNT-M A-DEPOSIT*1)). Those nodes in turn match METHODs for asking about PROPERTYs. The METHOD for ASK is one of the ports to the I/O model.

```
[(METHOD (ASK PROPERTY))
 OBJECT: <- (PROPERTY: DATUM):
 STEPS: (BECOME NEW-STATEMENT),
        (PRINT (STATEMENT (AND PLEASE_TYPE_IN_THE
                               OBJECT:))),
        (READ OBJECT:)]
```

It first declares that this is to be a new statement. That is, it should start on a new line. Next, it must print a request for the user to type in the desired information. STATEMENT is a module of the I/O model called by the evaluator that produces an appropriate statement. This statement is determined by looking at what the user thinks he is doing. In this case he is entering a deposit, therefore to request the (ACCOUNT A-DEPOSIT*1) it is only necessary to ask for the account. He will understand that it is of the deposit. Thus, the statement produced is "Please type in the account". Similarly for

the (AMOUNT-M A-DEPOSIT*1) only the AMOUNT-M need be asked for. Finally, the response is read from the user. The selection of a METHOD for printing calls a module that looks at the I/O situation before selecting a METHOD. In this case, the statement is to be on a new line and needs a space after it, therefore the Lisp PRINT is appropriate. There is a CODE-METHOD to set up the appropriate Lisp code, provided by the target language model. It also declares that it does I/O, providing order constraints for coding later. These order constraints are needed by the I/O model to maintain the proper console behavior of the program. The METHOD to read the ACCOUNT (or AMOUNT-M) does the reading, then tests the result to see that it agrees with the class requirements. The test for a number is the CODE-METHOD that applies the Lisp subroutine NUMBERP to it. Failing the test means that the procedure has failed. That is, an IF-FAIL property on the test has the value (FAILURE (METHOD (READ ...))). This failure is propagated up by analysis to a step that has a way to handle it, or if there are none it is handled as a failure exit from the program. In this case it will be a failure exit. An appropriate message to type out when this happens is found as:

[(FAIL-MESSAGE (TEST (IF-FAIL NUMBER))) <- ERROR_NOT_NUMBER]

For coding, the test and failure procedure are handled in the same way that an IF-THEN would be handled, building a COND that performs the test and doing the appropriate branching. This completes the ASK part of the ACCEPT programs.

The second step of the ACCEPT METHOD is not an ACTIVITY but an indication of a state change. It says the result returned by asking the user is now a datum. The handler for BECOME checks the superior node to see if anything should be done to reflect the existence of such a datum. On the superior node is the DURING property claiming it should be ASSERTed. To accomplish this, the BECOME is manifest as the following node:

```
[(ASSERT A-DEPOSIT*1)
 FOR: <- ((A-DEPOSIT*1 PROTO) (RESULT (ASK A-DEPOSIT*1)))]
```

That is, assert that the proto-a-deposit result of ASK is now an "a-deposit". The analysis of this node asks for the proto-a-deposit argument, which already exists, and sets up a goal to (IMPLEMENT A-DEPOSIT). That is, implement a data base in which it can be ASSERTed, before choosing a procedure to do the ASSERTing. That is all that can be done until the A-DEPOSITs and A-WITHDRAWALs are implemented.

Next, the METHODs for stating the balance are selected. As mentioned before, the top two nodes selected so far just produce the balance and then state it. The METHOD selected for (PRODUCE (BALANCE ACCOUNT*1)) is to (COMPUTE (DIFFERENCE ...)). This seems obvious after all of the analysis of the definition, but if it had been computed somewhere else since the last SOURCE, it could have been retrieved. The next goal is to select a METHOD for (COMPUTE (DIFFERENCE ...)), however to do that it is necessary to evaluate the arguments to the subtraction. Since the METHODs that return those arguments have not been selected yet, it is not possible to complete the selection and the goal is suspended. The effect is that the METHODs that make up the computation are chosen from the beginning of the argument chain working toward the final result. This would seem to suggest what might be happening when Brooks[2] noted the "design task nature" of coding (that the coding seems to proceed through the program one piece at a time with the consequences of each piece of code providing restrictions for the next piece) as exhibited in human programmers. In human terms they have mentally analyzed the situation sufficiently to start selecting programming language constructs for the lowest level operations, then work their way back through the imagined structure utilizing the known results of the constructs actually chosen.

---

[2] [Brooks 1975]

At this point the nodes awaiting METHODs to do the balance computation, listed in outline fashion, are as follows:

```
(COMPUTE (DIFFERENCE (PLUS ...)(SUM ...)))
   (COMPUTE (PLUS ((BALANCE ACCOUNT*1) INITIAL)(SUM ...)))
      (PRODUCE ((BALANCE ACCOUNT*1) INITIAL))
      (COMPUTE (SUM (SET (AMOUNT-M A-DEPOSIT*2))))
         <<(PRODUCE (SET (AMOUNT-M A-DEPOSIT*2)))>>
            (PRODUCE ACCOUNT*1)
            (PRODUCE (SET (AMOUNT-M A-DEPOSIT*3)))
   (COMPUTE (SUM (SET (AMOUNT-M A-WITHDRAWAL*2))))
      <<(PRODUCE (SET (AMOUNT-M A-WITHDRAWAL*2)))>>
         (PRODUCE ACCOUNT*1)
         (PRODUCE (SET (AMOUNT-M A-WITHDRAWAL*3)))
```

The two nodes in "<<>>" already have METHODs as a result of the characterization process, which provided the nodes following them.

Consider first the node to (PRODUCE ACCOUNT*1). Since the ACCOUNT is part of the identifier, the way to produce it is to ask the user. This is accomplished with the node (ASK ACCOUNT*1). The handling of this node is similar to the previous ASK node, except that ACCOUNT*1 is already a PROPERTY. In determining what to ask for, the I/O model must consider the relation of ACCOUNT*1 to the specification. The reason the ACCOUNT is needed is to specify which set of A-DEPOSITs must be produced, but the user knows nothing of that. He knows that he is running a program to give him the BALANCE of an ACCOUNT. Thus, the request to the user must be in terms of what he knows. Actually, it will just say "Please type in the account", but if further specificity were needed it would be "Please type in the account for which you want the balance". It should also be noticed that there are two nodes to produce the account, one for the set of deposits and one for the set of withdrawals. When the second METHOD selection goal is handled, the module (knowing that producing something is the kind of operation that only needs to be done once) searches for and finds the existing node to use.

The node to (PRODUCE (SET (AMOUNT-M A-DEPOSIT*3))) (and the corresponding A-WITHDRAWAL node) represents a need to have information whose SOURCE is in a different program execution. The way to do that is to RETRIEVE the information. Thus, the steps of the selected METHODs are (RETRIEVE (SET (AMOUNT-M A-DEPOSIT*3))) and (RETRIEVE (SET (AMOUNT-M A-WITHDRAWAL*3))) respectively. Analyzing these two nodes generates goals which now require:

```
[(IMPLEMENT A-DEPOSIT)
 FOR: <- (AND A-DEPOSIT*1
              A-DEPOSIT*3)] and
[(IMPLEMENT A-WITHDRAWAL)
 FOR: <- (AND A-WITHDRAWAL*1
              A-WITHDRAWAL*3)]
```

The node to (PRODUCE ((BALANCE ACCOUNT*1) INITIAL)) is simpler to handle than the others. As a result of the characterization it is known that the initial balance is zero and will not change. Therefore, the "method" simply returns zero.

Now that the terminal nodes of the computation have been handled, the next steps are the ones that do the computing. The first of these is (COMPUTE (SUM (SET (AMOUNT-M A-DEPOSIT*2)))). The METHOD for doing a sum on a set is an iteration through the elements, as follows:

```
[(METHOD (COMPUTE (SUM (SET NUMBER))))
 OBJECT: <- (SUM *SET=(SET NUMBER):)
 STEPS: (BECOME (VARIABLE: SOME) <- 0),
        [(ITERATE *SET)
         BODY:: <- (BECOME (VARIABLE: THE) <-
                           (PLUS (VALUE (VARIABLE: THE))
                                 (MEMBER *SET)))]
 RESULT: <- (VALUE (VARIABLE: THE))]
```

This METHOD has two steps, one which initializes a variable, and one which iterates through the set. The BODY property on the second step tells it what to do with the elements. The representation for objects being iterated over in the BODY is (MEMBER

*SET). Analyzing this METHOD sets up several goals. The two BECOMEs make changes to a VARIABLE, but since VARIABLEs are changed by ASSIGNing them, these are turned into calls to (ASSIGN VARIABLE*1). Since the assignment of variables is more easily handled at coding time the selection of suitable METHODs will be postponed. The argument to the first BECOME is no problem, since it is a constant. The argument to the second BECOME, however, is an arithmetic expression requiring an expansion, which takes place without complication. The analysis of (ITERATE (SET (AMOUNT-M A-DEPOSIT*2))) causes a goal to IMPLEMENT the set before selecting a METHOD.

Once the sums have been handled, the (COMPUTE (PLUS ...)) node is ready. Because the METHOD selection waited until the METHODs for the arguments were fully specified, it is possible to utilize everything known about the arguments. In this case one of the arguments will be zero, and there is a special METHOD for an addition of zero which just returns the other argument.

The last node is (COMPUTE (DIFFERENCE ...)). The METHOD for doing it is the obvious one of subtracting the two arguments. It should be noted that while both of the arguments have METHODs to produce them, there is no requirement for producing one before the other. This is proper. Such decisions will be left to the coding process.

The only part of (STATE ...) left is the actual printing of the value. The METHOD to STATE a DATUM determines that it is necessary to STATE the ACCOUNT and to STATE the VALUE. However, the METHOD selection module decides that it is not necessary to STATE the ACCOUNT, since it was just requested from the user. To STATE the VALUE there is a METHOD as follows:

```
[(METHOD (STATE (VALUE DATUM)))
 OBJECT: (VALUE DATUM:)
 STEPS: (BECOME NEW-STATEMENT),
        (PRINT (STATEMENT (AND THE (QUOTE (DATUM: THE))
                                IS (DATUM: THE))))]
```

(The QUOTE is a marker for EVALUATE to return the concept rather than the result form of the concept, i.e., (BALANCE ACCOUNT*1) rather than ((BALANCE ACCOUNT*1)(RESULT ...)).) The evaluation and METHOD selection procedures for the I/O model make use of the features of the situation and select CODE-METHODs to handle the printing. In this case it takes two steps, applying Lisp PRINT to "THE_BALANCE_IS_" and applying Lisp PRIN1 to the result of the computation.

At this point, all of the other goals are waiting for A-DEPOSIT and A-WITHDRAWAL to be implemented. (IMPLEMENT A-DEPOSIT) (A-WITHDRAWAL is analogous) means designing a data base to store the information and designing a form for the contents of the data. The data base will represent the (SET A-DEPOSIT), with the ITEMs in the data base representing the contents of the elements of the set. The operations on this set are to ASSERT new elements and to RETRIEVE the AMOUNT-Ms from the elements with the desired ACCOUNT. Because the retrieval of A-DEPOSITs is based on the ACCOUNT, it would simplify the retrieval process if the data base were indexed on the ACCOUNTs. In general it is desirable to have a data base indexed on the parts of the identifier that are NEEDED. That was one reason for doing the characterization of data.

Implementation is accomplished by putting the NEEDED identifiers in a list, ordering the list to make the required operations efficient, and creating levels of the data set to correspond to the elements of the list. In this case there is only one NEEDED identifier, therefore only one KEY, and therefore one corresponding level in the data base. Expanding the data base model in this situation produces the following correspondences between the data base parts and the set (the "->" indicates an implementation, which is actually declared in the design tree in a more complicated manner):

```
[DB-DEPOSIT
 NAME: <- DB-DEPOSIT
 DATA: -> (SET A-DEPOSIT)
 DATA-SECT: -> (SET A-DEPOSIT)
 KEY-DATA-SECT:
 KEY: -> ACCOUNT*2
 RECORD: -> (SET [A-DEPOSIT#6 ACCOUNT:: <- ACCOUNT*2])
 ITEM: -> [A-DEPOSIT*4 ACCOUNT:: <- ACCOUNT*2
                       ETIME:: <- TIME*1]
 (CONTENT ITEM:) -> (CONTENT A-DEPOSIT*4)]
```

The data base, DB-DEPOSIT, has a NAME and a DATA. The DATA is EQUIVALENT to its DATA-SECT, which is made up of KEY-DATA-SECTs (i.e. the KEY-DATA-SECTs are ELEMENTs of the DATA-SECT). These are pairs of KEY (the FIRST-PART of the KEY-DATA-SECT) and RECORD (the SECOND-PART of the KEY-DATA-SECT). Each KEY is an ACCOUNT of the deposits. Each RECORD has as ELEMENTs the ITEMs. The implementation of the (SET A-WITHDRAWAL) happens in an analogous way creating the data base DB-WITHDRAWAL.

Designing the data bases (i.e. IMPLEMENTing the sets as data bases) clears the way for selecting METHODs for handling the data, including the assertions and the retrievals for both the A-DEPOSITs and the A-WITHDRAWALs. Again, the A-DEPOSITs and A-WITHDRAWALs are analogous, so only the A-DEPOSITs will be discussed. The METHOD for (ASSERT A-DEPOSIT*1) becomes

```
[(INSERT (ITEM DB-DEPOSIT)*1)
 FOR: <- ((A-DEPOSIT*1 PROTO) (RESULT (ASK A-DEPOSIT*1)))]
```

as a result of the implementation of (SET A-DEPOSIT), because INSERT is a specialization of ASSERT for making one data base part an ELEMENT of another (this ITEM is an ELEMENT of the RECORD) and because A-DEPOSIT*1 became (ITEM DB-DEPOSIT)*1. The METHOD for (RETRIEVE (SET (AMOUNT-M A-DEPOSIT*3))) becomes (RETRIEVE [(RECORD DB-DEPOSIT)*1 ACCOUNT: <-ACCOUNT*1]) as a result of the implementation. Analyzing these two calls generates a goal that DB-DEPOSIT must be characterized.

Characterizing a data base is similar to the characterization done for the data, except that data bases are not inherently DEFINED the way sets are. Because the USEs of the data base could happen before the first SOURCE (there is no constraint on the relative times of execution), the data base must be marked (DEFINED NOT). The parts of the data base have characteristics corresponding to the characteristics of the parts of the set of data they represent. That means that all of the parts except the ITEM are DEFINED and CHANGE, the parts that are ELEMENTs are (KNOWN NOT), and the parts that are named parts (the parts of the DATA-BASE, and of the KEY-DATA-BASE) are KNOWN.

The next phase refines the insertion and retrieval algorithms for this situation. There are two kinds of ACTIVITYs involved: the data base level, which operate on the parts relative to the whole data base, and the Lisp level (indicated with an "L" prefix), which operate on the parts relative to the containing part. The METHOD for INSERTing an ITEM in a data base RETRIEVEs the RECORD, L-INSERTs the ITEM into the RECORD, and then if the result of the L-INSERT has the characteristic NEW it ASSERTs that it is the RECORD. If the RETRIEVE fails, it L-CREATEs the RECORD with the ITEM in it and ASSERTs the RECORD. The METHOD for the RETRIEVE checks to see if its OBJECT is EQUIVALENT to the DATA of the data base, and if so just L-RETRIEVEs the DATA. Otherwise, it RETRIEVEs the SUPERPART (containing part) and L-RETRIEVEs from that. The conditionals (IF-THEN in OWL-I) in these METHODs are handled in different ways depending on how much is known about the condition. If the condition is known, the appropriate branch is used. If the condition is known to vary, then a test can be selected and the conditional implemented as a Lisp conditional.

In the expansion of the INSERT routine, the first step is to RETRIEVE the

RECORD. There is an IF-FAIL property on it, so if RECORD is never going to exist the programwriter only needs to expand the property. However, there is no reason to believe that it will not exist since it corresponds to the set of A-DEPOSITs for the given ACCOUNT. The analysis of the conditional in RETRIEVE determines from the designed data base structure that the RECORD is not EQUIVALENT to the DATA. Hence, the KEY-DATA-BASE must be RETRIEVEd which is again not EQUIVALENT. Finally, the DATA-SECT must be RETRIEVEd which is EQUIVALENT. The structure so far (in outline form with the nodes numbered) is as follows:

```
<1>[(RETRIEVE *RECORD)]
  <2>[(RETRIEVE *KEY-DATA-SECT)]
    <3>[(RETRIEVE *DATA-SECT)]
      <4>[(L-RETRIEVE *DATA) FROM: <- DB-DEPOSIT]
    <5>[(L-RETRIEVE *KEY-DATA-SECT) FROM: <- (*DATA <3>)]
  <6>[(L-RETRIEVE *RECORD) FROM: <- (*KEY-DATA-SECT <2>)]
<7>IF-FAIL <1> ...
```

To answer the question of whether <1> can fail, it is necessary to look at <4>,<5>, and <6>, since all of these must succeed for <1> to succeed. Looking at <4>, the DATA is DEFINED and KNOWN, but the DATA-BASE is not DEFINED. The failure analysis concludes that it will not always fail, but can not determine whether it will ever fail because that depends on what kind of Lisp implementation is chosen. Looking at <5>, the KEY-DATA-SECT is not KNOWN so there is no way that it could always be available. Therefore, <5> can fail. Looking at <6>, the RECORD is KNOWN and DEFINED and its container, the KEY-DATA-SECT is given as an argument, so <6> can not fail. The result is that <1> can fail and both branches must be expanded. It should also be noted that if <5> fails, that means <2> fails and <1> fails without ever executing <6>, and similarly for <4>. Control after a failure goes up the node structure to the first place where it can be handled, which is the IF-FAIL. Since this IF-FAIL now indicates a branching in the control structure, it is necessary to set up a test to determine whether the node

succeeded. The node is (TEST (IF-FAIL <1>)). That will not be handled until the Lisp level implementations have taken place providing the information on what kind of test is needed. Continuing down the failure branch the expansion looks as follows:

```
<8>[(L-CREATE *RECORD) FROM: <- *ITEM]
<9>[(ASSERT *RECORD) FOR: <- (*RECORD <8>)]
  <10>[(RETRIEVE *KEY-DATA-SECT)]
  <11>IF-FAIL <10> ...
```

Again, a call to RETRIEVE the KEY-DATA-SECT is generated. This time the situation is different. It is known that <1> failed. However, <6> did not fail, so <2> must have failed also. Therefore <10>, which is the same as <2> must fail. Continuing:

```
<12>[(L-CREATE *KEY-DATA-SECT) FROM: <- (*RECORD <8>)]
<13>[(INSERT *KEY-DATA-SECT) FOR: <- (*KEY-DATA-SECT <12>)]
  <14>[(RETRIEVE *DATA-SECT)]
  <15>IF-FAIL <14> ...
```

This time the problem is the same as for determining whether <4> will fail. It must wait until the DATA-BASE is implemented as a Lisp structure. Going back to the branch where <1> does not fail, the next steps are as follows:

```
<16>[(L-INSERT *ITEM) FOR: <- A-DEPOSIT*3 IN: <- (*RECORD <1>)]
<17>IF-THEN (CHARACTERIZED <16> NEW) ...
```

Again, there is a conditional that must be determined before the branches can be expanded with certainty. Being characterized as NEW means that the result of the L-INSERT is an entity that might no longer be in its container. That is, the resulting RECORD must be placed back in the KEY-DATA-SECT. The answer to this question depends on the Lisp structures and operations used for the L-INSERT.

In the STATE program the (RETRIEVE *RECORD) for producing the set of A-DEPOSITs is the same as the RECORD retrieval above. If this retrieval fails, the PRODUCE catches it and substitutes the DEFINED value, which is an empty RECORD. The implementation has also changed the iteration to (ITERATE (*RECORD (RESULT ...))). The

(MEMBER (SET ...)), which is an ITEM, becomes the (MEMBER *RECORD) which is also the ITEM, causing no problems of too much structure. Again however, to continue any farther, it is necessary (according to the INTENT for (ITERATE RECORD)) to decide on an implementation of the RECORD at the Lisp level.

At this point, everything is waiting on the implementation of the data base parts in Lisp. This is the main entrance point of the target language model. It now has the responsibility of fitting the constructs that have been designed into Lisp. Such implementations depend mostly on the relationships between the object and its contents and the way the contents are used. The DATA-BASE is a global entity having a NAME and the DATA. There are two kinds of global Lisp entities that the programwriter knows about, symbols (i.e., Lisp free variables) and the properties on property lists. Let us assume the programwriter chooses a property on a property list (a PL-PROPERTY). It is implemented with the NAME of the DATA-BASE used as the property list name (i.e., the symbol having the property list, or PL-NAME in Owl-I) and the symbol DATA-BASE as the indicator on the property list (the PL-INDICATOR). A PL-PROPERTY automatically has an initial value of NIL, a fact which is indicated on the concept. The DATA-SECT is made up of a set of pairs, and the KEY part of each pair is used for retrieval. This is exactly the situation needed for the use of association lists. The SCHEMA for ASSOCIATION-LISTs turns the DATA-SECT into an ASSOCIATION-LIST of the KEY-DATA-SECTs and turns the KEY-DATA-SECTs into DOTTED-PAIRs of KEY and RECORD. Thus, retrieval of a KEY-DATA-SECT is done with the Lisp ASSOC applied to the DATA-SECT and the desired KEY. As mentioned, an empty DATA-SECT is an empty ASSOCIATION-LIST, which is an empty LIST, which is NIL. The RECORD, the SECOND-PART of the DOTTED-PAIR, is implemented as a LIST because it is made up of a variable number of ITEMs. Each of the ELEMENTs of the LIST will represent one of the ITEMs, and an empty LIST, NIL, will

represent an empty RECORD, and hence an empty SET. Thus, the implementations are as follows:

```
[PL-PROPERTY -> DB-DEPOSIT
  PL-NAME: <- DB-DEPOSIT
  PL-INDICATOR: <- DATA-BASE
  VALUE: -> DATA]
[ASSOCIATION-LIST -> DATA
  ELEMENT: -> KEY-DATA-SECT]
[DOTTED-PAIR*1 -> [KEY-DATA-SECT*1  FIRST-PART: <- KEY*1]
  FIRST-PART: -> KEY*1
  SECOND-PART: -> RECORD*1]
[LIST*1 -> RECORD*1
  ELEMENT: -> ITEM*1]
```

The new implementations are added to the structure in the same way as the data base implementations, causing the call to L-RETRIEVE the RECORD to become a call to L-RETRIEVE the SECOND-PART of the DOTTED-PAIR, causing the call to L-INSERT into the RECORD to become a call to L-INSERT into a LIST, and so forth. These in turn require the characterizations of the PL-PROPERTY, the ASSOCIATION-LIST and the LIST. The characterizations are easy to do because all of the needed information is declared on the Lisp structures or inherited from the data structures that were implemented. The characterization determines that the PL-PROPERTY is DEFINED, having an initial value of NIL and that the ASSOCIATION-LIST is EQUIVALENT to the (VALUE PL-PROPERTY). The ASSOCIATION-LIST and the LISTs for the RECORDs are also DEFINED with initial values of NIL. Each DOTTED-PAIR is DEFINED with the FIRST-PART initially the ACCOUNT (which is the KEY, which is a NUMBER) and the SECOND-PART initially the initial RECORD.

At this point the programwriter can continue the expansion of the data base, suspended before for lack of information. Since the PL-PROPERTY is DEFINED, the value returned from an initial retrieval of the VALUE must either uniquely indicate failure, be the correct initial value, or there must be some other way of detecting error, or an

initialization program must be proposed. As it turns out, the initial VALUE of the PL-PROPERTY is NIL; the (VALUE PL-PROPERTY) represents the ASSOCIATION-LIST, which has an initial VALUE of NIL; and the initial SET is empty, which means the DATA is empty, which means the ASSOCIATION-LIST should be empty, which is represented by NIL. Given these fortuitous circumstances, <15> (i.e., IF-FAIL (RETRIEVE *DATA-SECT) ...) can not fail. Thus, following <15> is the non-failure branch only:

```
<18>[(L-INSERT *KEY-DATA-SECT)
     FOR: <- (*KEY-DATA-SECT <12>)
     IN: <- (*DATA-SECT <14>)]
<19>IF-THEN (CHARACTERIZED <18> NEW) ...
```

This time it is possible to tell whether the result will be NEW. Since the *DATA-SECT is implemented as *ASSOCIATION-LIST and *KEY-DATA-SECT is implemented as *DOTTED-PAIR, <18> becomes:

```
[(L-INSERT *DOTTED-PAIR)
 FOR: <- (*DOTTED-PAIR <12>)
 IN: <- (*ASSOCIATION-LIST <14>)]
```

There is a CODE-METHOD for the L-INSERT as follows:

```
[(CODE-METHOD (L-INSERT LISP-ENTITY))
 OBJECT: <- [LISP-ENTITY:
             (ELEMENT LIST:)]
 IN: <- LIST:
 FOR: <- (LISP-ENTITY: PROTO)
 STEPS: (LISP CONS FOR: IN:)
 [RESULT: LIST NEW]]
```

This CODE-METHOD uses the Lisp primitive CONS to include the new element in the DATA-SECT and it has the characteristic NEW on the RESULT. Thus, it is necessary to continue expansion of the TRUE branch after <19> as follows:

```
<20>[(ASSERT *DATA-SECT) FOR: <- (*DATA-SECT <18>)]
  <21>[(L-ASSERT *DATA) FOR: <- (*DATA-SECT <18>)
                        IN: <- DB-DEPOSIT]
```

The other branch where <1>'s retrieval of the *RECORD had succeeded was suspended awaiting information about the properties of the result of <16>. That L-INSERT also

matches the CODE-METHOD above, making the result NEW. Thus, the expansion following <17> is for the TRUE branch:

```
<22>[(ASSERT *RECORD) FOR: <- (*RECORD <16>)]
  <23>[(RETRIEVE *KEY-DATA-SECT)]
  <24>[(L-ASSERT *RECORD) FOR: <- (*RECORD <16>)
                          IN: <- (*KEY-DATA-SECT <23>)]
  <25>IF-THEN (CHARACTERIZED <24> NEW) ...
```

Again, it is necessary to know if the result of the L-ASSERT will be NEW. In this case the CODE-METHOD that matches uses the Lisp primitive RPLACD to make the RECORD the SECOND-PART of the KEY-DATA-SECT. This does not result in a NEW KEY-DATA-SECT, so this is the end of the expansion.

Now, the rest of the programming falls into place. Each of the Lisp level calls (the ones starting with "L-") has a CODE-METHOD that provides the necessary Lisp code for the operation. The procedure for (L-RETRIEVE PL-PROPERTY) has a CODE-METHOD that does a Lisp GET of the appropriate arguments,

[(L-RETRIEVE DOTTED-PAIR) FROM: <- ASSOCIATION-LIST]

uses Lisp ASSOC, and so forth. For (ITERATE LIST*) there exists a CODE-METHOD that does a Lisp DO[3]. To set up the iteration as a DO the BODY must be included in the DO. The DO sets up the iteration by initializing a variable to the LIST (the implementation of the RECORD), and resetting it to the rest of the LIST after each time through the body. This means that (MEMBER LIST) in the body must be handled as (CAR -variable-). The DO ends when the list is empty, tested for by the Lisp predicate NULL.

Once all of the goals are completed and all of the nodes end in CODE-METHODs (except for the ASSIGNs of variables, which were specifically suspended awaiting the coding), the analysis and planning phase is over. The only thing left to do is

---

[3] A MacLisp DO, of the form (DO (initialization of variables)(end test, end form) body), see [Moon, 1974]

to generate the code from the information gathered on the structure. The coding section is the major module of the target language model and has to decide how to handle the passing of arguments, select orderings not already constrained, control the use of local variables and handle the details of quoting and program structure. These tasks are fairly mechanical to handle but of a somewhat different flavor than the refinement process. One of the tasks is to handle the local variables. Most of these will occur as the result of needing arguments in more than one place, but in the summation METHOD a variable was explicitly used to accumulate the sum. Presently, [(ASSIGN VARIABLE*1) VALUE: <- 0] is constrained to come before the DO and (ASSIGN VARIABLE*1) and (VALUE VARIABLE*1) are used in the body of the DO. This is a situation where the variable can be included as part of the DO. It is a situation that the coding routine recognizes and turns into an initialization assignment of the DO. After handling the various coding tasks, the resulting set of programs are ready to handle the user's task specification. The variables have been given mnemonic names to assist the reader.

```
(DEFUN ACCEPT-DEPOSIT ()
  (PROG (ACCT AMNT DATA KDS)
        (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
        (SETQ ACCT (READ))
        (COND ((NUMBERP ACCT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (PRINT 'PLEASE_TYPE_IN_THE_AMOUNT_OF_MONEY)
        (SETQ AMNT (READ))
        (COND ((NUMBERP AMNT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (SETQ DATA (GET 'DB-DEPOSIT 'DATA-BASE))
        (SETQ KDS (ASSOC ACCT DATA))
        (COND (KDS (RPLACD KDS (CONS AMNT (CDR KDS))))
              (T (PUTPROP 'DB-DEPOSIT
                          (CONS (CONS ACCT (LIST AMNT)) DATA)
                          'DATA-BASE)))
        (RETURN 'THANK_YOU)))
```

```
(DEFUN ACCEPT-WITHDRAWAL ()
  (PROG (ACCT AMNT DATA KDS)
        (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
        (SETQ ACCT (READ))
        (COND ((NUMBERP ACCT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (PRINT 'PLEASE_TYPE_IN_THE_AMOUNT_OF_MONEY)
        (SETQ AMNT (READ))
        (COND ((NUMBERP AMNT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (SETQ DATA (GET 'DB-WITHDRAWAL 'DATA-BASE))
        (SETQ KDS (ASSOC ACCT DATA))
        (COND (KDS (RPLACD KDS (CONS AMNT (CDR KDS))))
              (T (PUTPROP 'DB-WITHDRAWAL
                          (CONS (CONS ACCT (LIST AMNT)) DATA)
                          'DATA-BASE)))
        (RETURN 'THANK_YOU)))

(DEFUN STATE-BALANCE ()
  (PROG (ACCT KDSD KDSW)
        (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
        (SETQ ACCT (READ))
        (COND ((NUMBERP ACCT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (PRINT 'THE_BALANCE_IS_)
        (SETQ KDSD (ASSOC ACCT
                          (GET 'DB-DEPOSIT 'DATA-BASE)))
        (SETQ KDSW (ASSOC ACCT
                          (GET 'DB-WITHDRAWAL 'DATA-BASE)))
        (PRIN1 (DIFFERENCE
                 (DO ((LST (COND (KDSD (CDR KDSD))
                                 (T NIL))
                           (CDR LST))
                      (SUMD 0))
                     ((NULL LST) SUMD)
                     (SETQ SUMD (PLUS SUMD (CAR LST))))
                 (DO ((LST (COND (KDSW (CDR KDSW))
                                 (T NIL))
                           (CDR LST))
                      (SUMW 0))
                     ((NULL LST) SUMW)
                     (SETQ SUMW (PLUS SUMW (CAR LST))))))
        (RETURN 'THANK_YOU)))
```

Section 2 Introducing Ideas for Efficiency

The programs that have been produced are not very efficient. They require ever increasing amounts of storage space and ever increasing amounts of time the more they are used. Yet, these were what resulted from the simple specifications and applying the definition of balance directly. It is possible to improve on these programs if the programwriter makes use of the IDEAs that the analyzer picks up along the way. IDEAs are similar to GOALs in form but there is no requirement that they be used. The analyzer looks for IDEAs on the structures it finds and adds them to an IDEA list. They are tested by comparing their results to the results without the IDEA at as high a level as possible. That means being able to construct hypothetical nodes and having some measure on which to compare the nodes.

To see how IDEAs work, consider what happens when BALANCE is characterized as TOTAL. The analyzer conducts a search for IDEAs on each node. When it searches from (PRODUCE (BALANCE ACCOUNT*1)) it finds the concept (IDEA (PRODUCE BALANCE)). This is not actually an IDEA, but it does point to one (e.g., have one on its reference list). The IDEA it points to is:

```
[(DEFINITION (IDEA (PRODUCE TOTAL)))
 OBJECT: <- (IDEA (PRODUCE TOTAL:))
 RESULT: <- [(IMPLEMENT (PRODUCE (TOTAL: THE)))
           AS: <- (SUBTOTAL (TOTAL: THE))]]
```

The reason this idea was not found directly is the SUBSTANTIVE-CHARACTERIZATION in the specializer corresponding to the position of (BALANCE ACCOUNT*1). To verify that it really applies, (BALANCE ACCOUNT*1) is checked for the characteristic TOTAL, which is found. To SUBTOTAL something means to use the result of the previous such computation as the initial value for the current computation. This IDEA is evaluated and put on the IDEA list as

```
[(IMPLEMENT (PRODUCE (BALANCE ACCOUNT*1)))
 AS: <- (SUBTOTAL (BALANCE ACCOUNT*1))].
```

A goal is also set up to (CHECK IDEAS), which happens just before any IMPLEMENT goals.

To handle an IDEA, it is useful to set up an EVENT in the programwriter describing the proceedings as HYPOTHETICAL. This permits easy validating and invalidating of the results, making it possible to compare the results of different IDEAs and the program without IDEAs. This EVENT is attached to the things created while investigating the IDEA, as part of the normal binding and node formation mechanism. Once the EVENT exists, handling the IDEA is the same as handling a goal.

The SCHEMA for the implementation of (SUBTOTAL AMOUNT) has two parts, turn the (PRODUCE (AMOUNT: THE)) into (SUBTOTAL (AMOUNT: THE)) and if any of the SOURCEs of the AMOUNT: have ETIMEs less than the ETIME of the program execution they are in (i.e., it is a change or addition of an old value), FIX-UP the AMOUNT: at the places with such ETIMEs. That is, change the AMOUNT: to reflect the change in one of its arguments. In this case the SOURCEs are all from ACCEPT programs, requiring the data to have the same ETIME as the program execution, so there is no problem with "fixing up". The PRODUCE is just turned into a call to SUBTOTAL. The interesting part comes when the steps of the METHOD for SUBTOTAL are analyzed. The METHOD is as follows:

```
[(METHOD (SUBTOTAL AMOUNT))
 OBJECT: <- AMOUNT:
 STEPS: [*COMP=(COMPUTE (AMOUNT: THE))
        KNOW:: <- [(ETIME (ULTIMATE-ARG:: THE))
                   (SINCE (SUBTOTAL (AMOUNT: THE)))]],
       (BECOME (AMOUNT: INITIAL) <- (RESULT *COMP))]
```

This METHOD will do the "subtotaling" of (BALANCE ACCOUNT*1). The first step in it is to compute the amount, which in this case is the (BALANCE ACCOUNT*1). Notice that the METHOD is not dependent on the fact that the computation is for a balance.

The analysis of the computation is the same as the analysis of the definition expansion in the first scenario, with a few exceptions. First, the ULTIMATE-ARGs to be used in the computation are only those that have been created since the previous time the amount was subtotaled. The property KNOW means that its value is known to be true of the specializer and the design should proceed utilizing that knowledge. Secondly, a different METHOD will be chosen for the PLUS because characterization will show that the initial BALANCE is no longer always zero. After the computation takes place the result becomes the new value for the initial amount. Thirdly, analyzing (SUBTOTAL (BALANCE ACCOUNT*1)) determines that it is a SOURCE for ((BALANCE ACCOUNT*1) INITIAL), which will change the characterization of the BALANCE. It also discovers another IDEA, to implement SUBTOTAL as INCREMENTAL-TOTAL, which will be explored later. Finally, the analysis of the DEFINITION picks up the ULTIMATE-ARG restrictions by verifying that ((BALANCE ACCOUNT*1) INITIAL) is a legitimate argument to part of the computation and restricting the sets to

```
(SET (AMOUNT-M [A-DEPOSIT*5 PATTERN
            ACCOUNT: <- ACCOUNT*1
            [ETIME: (SINCE (ACT (SUBTOTAL
                              (BALANCE ACCOUNT*1))))]]))
and
(SET (AMOUNT-M [A-WITHDRAWAL*5 PATTERN
            ACCOUNT: <- ACCOUNT*1
            [ETIME: (SINCE (ACT (SUBTOTAL
                              (BALANCE ACCOUNT*1))))]])),
```

respectively.

Characterizing the BALANCE determines that the SOURCE for ((BALANCE ACCOUNT*1) INITIAL) is now (STATE ...), that it CHANGEs, is a single item, and is still DEFINED initially as zero. Also, (BALANCE ACCOUNT*1) still is DEFINED, CHANGEs, and is a single item, but its SOURCE is now the previous (STATE ...) and all of the (ACCEPT ...) executions since the (STATE ...). The characterization of the A-DEPOSITs and A-WITHDRAWALs and their sets are essentially the same as before.

Once the analysis and planning of the IDEA gets to the stage of implementing, it will be time to check whether all of this was worthwhile. Before that happens, there is another IDEA needing to be checked. This one will turn the SUBTOTAL into an INCREMENTAL-TOTAL. That means the AMOUNT-M is computed at the SOURCE each time and ASSERTed so it can be RETRIEVEd when it is used. The SCHEMA consists of four steps, turn the (SUBTOTAL ...) into (RETRIEVE ...) and then the following:

```
[(DO *SUBT=(SUBTOTAL AMOUNT:),
    (BECOME (AMOUNT: THE) <- (RESULT *SUBT)))
 LOCATION:: <- (AFTER [(SOURCE *ARG=(ULTIMATE-ARG (AMOUNT: THE)))#1
                       ((ETIME ACT::) <- (ETIME *ARG))])]
```

This puts the SUBTOTAL procedure into the (ACCEPT A-DEPOSIT*1) and (ACCEPT A-WITHDRAWAL*1) programs after they have become the SOURCEs for the ULTIMATE-ARGs of BALANCE, that is, after the BECOME. The ETIMEs of the A-DEPOSITs and A-WITHDRAWALs are the same as the ETIMEs of the ACCEPT programs, satisfying the conditions. The third step is in case the SOURCEs have ETIMEs different from the data they produce, which is not the case here. It is not needed this time, but will be encountered in a later scenario. Finally, the last step is to (ANALYZE (SOURCE (AMOUNT: THE))), which becomes a goal to redo the analysis of the SOURCEs.

This IDEA introduces a few new wrinkles. Characterizing the

(SET (AMOUNT-M [A-DEPOSIT#7 [ETIME:: (SINCE (SUBTOTAL ...))]]))

in (ACCEPT A-DEPOSIT*1) produces the fact that there is only one such amount, the amount of the A-DEPOSIT*1 just asserted. In (ACCEPT A-WITHDRAWAL*1) the set of such A-DEPOSITs is empty. This is discovered because SUBTOTAL happens in (ACCEPT A-DEPOSIT*1) (and also (ACCEPT A-WITHDRAWAL*1)) after the transactions are ASSERTed, which eliminates all previous SOURCEs of A-DEPOSIT. Characterizing the (BALANCE ACCOUNT*1) results in ((BALANCE ACCOUNT*1) INITIAL) being equated with

(BALANCE ACCOUNT*1). That is, they have the same SOURCEs now, so only one needs to be ASSERTed.

When it comes to selecting METHODs for the sums, there is a special METHOD for the sum of a set with one element, that consists of simply returning the element. There is also a METHOD for the sum of an empty set that returns zero, a METHOD for PLUS of zero and a number that returns the number, and a METHOD for the DIFFERENCE of a number and zero that returns the number. After selecting the appropriate METHODs, the subtotal in (ACCEPT A-DEPOSIT*1) is reduced to (PLUS (BALANCE ACCOUNT*1) (AMOUNT-M (RESULT ...))) and in (ACCEPT A-WITHDRAWAL*1) to (DIFFERENCE (BALANCE ACCOUNT*1) (AMOUNT-M (RESULT ...))).

Reanalyzing the SOURCEs of the A-DEPOSITs and A-WITHDRAWALs means looking at the BECOME statements again. This time there are no USEs of the A-DEPOSITs and A-WITHDRAWALs outside of the program they occur in, so there is no reason to do an ASSERT.

Now that the ideas have been handled, it is necessary to see whether the results are worth keeping. To do this some metric is needed. The metric that will be used is a very simple one, comparing the size of the sets used and secondarily comparing the computations. There are three classes of sizes: constant, sawtooth, and increasing, each much worse than the preceding. The sets and computations in the highest category are compared with each other. The sets are considered to have as many "dimensions" as identifiers and must be compared along each compatible dimension. It may be noticed that in fact the reading and printing on the terminal take more time than anything else, but that is considered necessary time not to be compared.

Consider first the original METHOD of computing the sum. There are two sets of data used, (SET A-DEPOSIT) and (SET A-WITHDRAWAL). In the ACCOUNT dimension these sets are considered (CONSTANT APPROX), because there is information provided by the domain model that the set of accounts is approximately constant. In the ETIME dimension on the other hand, the sets are INCREASING, because the RATE of ACTs of depositing to any given account is considered to be (CONSTANT APPROX). The only computations in the programs that will require more than constant time are the two sums (determined by looking for iterations), which are proportional to the size of the sets in the ETIME dimension and hence increasing. In the subtotal method of computing the sum, the sets of data used are

```
(SET (AMOUNT-M [A-DEPOSIT#7
            [ACCOUNT: ACCOUNT#2]
            [ETIME: (SINCE (ACT (SUBTOTAL
                              (BALANCE ACCOUNT#2))))]]))
and
(SET (AMOUNT-M [A-WITHDRAWAL#7
            [ACCOUNT: ACCOUNT#2]
            [ETIME: (SINCE (ACT (SUBTOTAL
                              (BALANCE ACCOUNT#2))))]]))
and
(SET ((BALANCE ACCOUNT#3) INITIAL)).
```

All of these are still proportional to the size of the set of ACCOUNTs in the ACCOUNT dimension. In the ETIME dimension the first two have changed. They are now SAWTOOTH, meaning that there is some cutoff relative to a recurring event or a time measured relative to the current time. The computation of the sum is again the only part of the programs requiring more than constant time, and it is proportional to the set for a given ACCOUNT. Since the increasing nature of the sizes relative to ETIME has been removed by the subtotal IDEA, it is better than the original.

Comparing the SUBTOTAL IDEA to the INCREMENTAL-TOTAL IDEA is done in the same way. The only data used by the programs is the (BALANCE ACCOUNT#1),

which is of constant size in ETIME (only one) and (CONSTANT APPROX) size in ACCOUNT (the size of the set of ACCOUNTs again). Also, the computations have all been reduced to constant size. Therefore, the INCREMENTAL-TOTAL IDEA is better than the SUBTOTAL IDEA. Thus, it is accepted and the hypothetical events created to represent the handling of the IDEA are marked ACTUAL. This includes both the event for the SUBTOTAL IDEA and the event for the INCREMENTAL-TOTAL. The programwriter is not confused by all of these overlapping values and characterizations because it uses whichever value is the result of the most recent ACTUAL event. This completes the checking of IDEAs, leaving the implementation goals, some of which will be dispensed with because they no longer apply.

From here on, the processing is similar to the original programs but simpler. The data base implemented for (BALANCE ACCOUNT*1) has the RECORD EQUIVALENT to the ITEM, which is implemented as a PL-PROPERTY.

The final set of programs is as follows:

```
(DEFUN ACCEPT-DEPOSIT ()
  (PROG (ACCT AMNT DATA KDS NBAL)
        (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
        (SETQ ACCT (READ))
        (COND ((NUMBERP ACCT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (PRINT 'PLEASE_TYPE_IN_THE_AMOUNT_OF_MONEY)
        (SETQ AMNT (READ))
        (COND ((NUMBERP AMNT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (SETQ DATA (GET 'DB-BALANCE 'DATA-BASE))
        (SETQ KDS (ASSOC ACCT DATA))
        (SETQ NBAL (PLUS AMNT (COND (KDS (CDR KDS))
                                    (T 0))))
        (COND (KDS (RPLACD KDS NBAL))
              (T (PUTPROP 'DB-BALANCE
                          (CONS (CONS ACCT NBAL) DATA)
                          'DATA-BASE)))
        (RETURN 'THANK_YOU)))
```

```
(DEFUN ACCEPT-WITHDRAWAL ()
  (PROG (ACCT AMNT DATA KDS NBAL)
        (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
        (SETQ ACCT (READ))
        (COND ((NUMBERP ACCT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (PRINT 'PLEASE_TYPE_IN_THE_AMOUNT_OF_MONEY)
        (SETQ AMNT (READ))
        (COND ((NUMBERP AMNT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (SETQ DATA (GET 'DB-BALANCE 'DATA-BASE))
        (SETQ KDS (ASSOC ACCT DATA))
        (SETQ NBAL (DIFFERENCE (COND (KDS (CDR KDS))
                                     (T 0))
                               AMNT))
        (COND (KDS (RPLACD KDS NBAL))
              (T (PUTPROP 'DB-BALANCE
                          (CONS (CONS ACCT NBAL) DATA)
                          'DATA-BASE)))
        (RETURN 'THANK_YOU)))

(DEFUN STATE-BALANCE ()
  (PROG (ACCT KDS)
        (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
        (SETQ ACCT (READ))
        (COND ((NUMBERP ACCT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (SETQ KDS (ASSOC (GET 'DB-BALANCE 'DATA-BASE)))
        (COND (KDS (CDR KDS))
              (T 0))
        (RETURN 'THANK_YOU)))
```

If the INCREMENTAL-TOTAL IDEA had not succeeded and only the SUBTOTAL IDEA had been used, the processing would also have been similar. The interesting part for this case is how to implement the set of A-DEPOSITs (or A-WITHDRAWALs).

```
(SET [A-DEPOSIT#1
      ETIME: (SINCE (ETIME (ACT (SUBTOTAL
                                 (BALANCE ACCOUNT:))))])
```

This set is the subset of the A-DEPOSITs that have been received since the last running of the subtotal program for any particular account. The characterization of the set recognizes the form

```
(SET [DATUM#1 ETIME: (SINCE (ETIME ACT#1))])
```

as implementable without recourse to a requirement for recording the times of the activity. The characterization marks the SINCE restriction as handled and sets up a goal for a PREMETHOD to include the following call:

```
[(PRUNE (SET *DATUM=A-DEPOSIT*1))
 (AFTER *ACT=(SUBTOTAL (BALANCE ACCOUNT)))]
```

This clears the data base after the subtotaling has been done, making it ready to take on the new A-DEPOSITs or A-WITHDRAWALs. Implementing the PRUNE is also fairly easy because it is equivalent to ASSERTing an empty RECORD for that ACCOUNT. Ultimately it just becomes (RPLACD *DOTTED-PAIR* NIL) where *DOTTED-PAIR* is the KEY-DATA-SECT that has the RECORD.

Section 3 A Variation on the Specification

To understand how the restrictions in the specifications were used and the effects they had on the resulting program, it is useful to consider alternatives to them. In this scenario the programwriter will handle a specification with the same requests as before plus two additional requests. One of the additional programs is to make corrections to deposits submitted within the last month, and one is to make corrections to withdrawals submitted within the last month. The specification for the program to correct deposits is as follows:

```
[(CORRECT *DEP=A-DEPOSIT*1)
 REQUIRE: <- (> (ETIME *DEP)(DIFFERENCE (ETIME ACT:) 1-MONTH))]
```

The INTENT for CORRECT is the same as the INTENT for ACCEPT except that CORRECT has a REQUIRE that the ETIME of the DATUM be less than the ETIME of the CORRECT (i.e., that it be a prior time), instead of the KNOW property that is on the ACCEPT's INTENT. The two REQUIRE properties on the request and the INTENT imply that both

tests must be passed for an execution of the program to complete successfully. If the test is not passed, the program should return an appropriate error message.

The analysis of the CORRECT programs proceeds in the same way as the analysis of the ACCEPT programs, except that the two REQUIREs must be analyzed. Both are simple predicates taking two arguments. The (ETIME A-DEPOSIT*1) is part of the identifier of A-DEPOSIT*1 and readily available. The (ETIME (ACT (CORRECT A-DEPOSIT*1))) is the time when the program takes place, which is available by calling the primitive TODAY. The interesting differences occur when the data is to be characterized. This time there are two SOURCEs for the A-DEPOSITs, and the CORRECT SOURCE will admit A-DEPOSITs that have already been admitted. Therefore, A-DEPOSITs can change and the SOURCE in CORRECT will also have to be considered a USE of the A-DEPOSITs. On the other hand, if the ETIME of an A-DEPOSIT is more than a month old it can no longer change. To indicate this situation, the A-DEPOSITs are given a CHANGE property with the time of execution restriction on it. Because of the CHANGE property, the needed identifiers of the A-DEPOSITs include the whole identifier set, which is the ACCOUNT and the ETIME.

The METHODs for asking the user will produce different results for the CORRECT program than for the ACCEPT program. The ETIME of the A-DEPOSIT in the ACCEPT program is available because from the KNOW property in the INTENT for ACCEPT it is the ETIME of the program execution. In the CORRECT program, however, it is not known and is needed for testing the requirements and for one of the USEs. Thus, it must be asked for. Since it is part of the identifier, the (ASK (ETIME ...)) call is placed along with the (ASK (ACCOUNT ...)) before the (ASK (AMOUNT-M ...)). Therefore, the two parts of the identifier are asked for before the contents.

Implementing the A-DEPOSITs also differs from the original scenario, because now the whole identifier is needed. Both parts of the identifier must be used as KEYs so the A-DEPOSITs can be retrieved for correction. With more than one KEY in the KEY-LIST it is necessary to decide on an ordering for the levels. The choice is made by checking the retrieval patterns and localizing the retrievals as much as possible. In this case the STATE programs will need to RETRIEVE the (SET (AMOUNT-M [A-DEPOSIT*1 ACCOUNT: <- ACCOUNT*1])), so the ACCOUNT should come first. The resulting data base has the following structure:

```
DATA
EQUIVALENT DATA-SECT*1
        ELEMENT KEY-DATA-SECT*1
             FIRST-PART [KEY*1 ACCOUNT]
             SECOND-PART DATA-SECT*2
                    ELEMENT KEY-DATA-SECT*2
                         FIRST-PART [KEY*2 ETIME]
                         SECOND-PART RECORD
                                EQUIVALENT ITEM
```

This data base in turn changes the implementation of the data base parts into Lisp. But first, consider what would happen if the data base itself were implemented as a Lisp free variable (symbol) instead of a property list entry. Again the failure analysis was invoked and suspended when it was discovered that the data base might be accessed before it was defined. This time when it is resumed at the Lisp level, accessing the variable can produce an error because the value of a Lisp variable is not initially defined, but the error situation can be detected by using the Lisp subroutine BOUNDP before accessing the variable. This means that the failure branch of the program at that point would have to be expanded, which includes a call to (L-CREATE *DATA) and then to (L-ASSERT (*DATA ...)). On the other hand, the variable could be initialized in the beginning, avoiding doing the BOUNDP each time if there is a legitimate value that it can be initialized to. The DATA is implemented as an ASSOCIATION-LIST

and is initially empty, so the initial value will be NIL. The programwriter then makes up an additional program to be run once in the beginning, called INITIALIZATION, that sets the variable to NIL.

Each of the KEY-DATA-SECTs is implemented as a DOTTED-PAIR. The two DATA-SECTs are implemented as ASSOCIATION-LISTs. However, the RECORD is EQUIVALENT to the ITEM, meaning it is not separately implemented.

The summation of the A-DEPOSITs produces different code with this configuration. The sum of the SET is the sum of the DATA, which is the sum of the ITEMs in the DATA-SECT*2. The METHOD for summing a SET provides the same (ITERATE (SET ...)) as before, but when this becomes (ITERATE DATA-SECT*2) there is a problem. The ELEMENT of DATA-SECT*2 is KEY-DATA-SECT*2, but the MEMBER that is needed is the ITEM. The METHOD for [(ITERATE DATA-SECT) FOR: ITEM] has a BODY of its own. It L-RETRIEVEs the SUBPART of the KEY-DATA-SECT (either a DATA-SECT or a RECORD). If that part is a kind of ITEM (if it is a RECORD that is EQUIVALENT to the ITEM), the BODY passed in by the call is applied to it. Otherwise, the iteration is done in turn on the SUBPART utilizing the BODY that was passed in. In this case the KEY-DATA-SECT SUBPART is an ITEM (because of the EQUIVALENT), so L-RETRIEVEing it for the BODY is all that is necessary. Thus, the BODY becomes:

```
<1>[(L-RETRIEVE (ITEM ...))
    FROM: <- (KEY-DATA-SECT*2 (ELEMENT DATA-SECT*2))],
<2>[(ASSIGN VARIABLE*1)
    FROM:  <- (COMPUTE (PLUS (VALUE VARIABLE*1)
                             (ITEM <1>)))]
```

Since the KEY-DATA-SECT*2 is implemented as a DOTTED-PAIR, the only change this makes in the original program is to change

```
(SETQ SUMD (PLUS SUMD (CAR LST))) to
(SETQ SUMD (PLUS SUMD (CDR (CAR LST))))
```

and similarly for SUMW (sum of the withdrawals).

Asserting an A-DEPOSIT in both the CORRECT program and the ACCEPT program becomes a little more complicated. The addition of another level means that retrieval, creation, and assertion will involve more steps. The CORRECT program requires the same decisions as the ACCEPT program plus a few more, so only the CORRECT program will be discussed. The first step of ASSERT in this situation is to (RETRIEVE KEY-DATA-SECT*2) (the SUPERPART of the ITEM in this case), but this can fail. This corresponds to (RETRIEVE *RECORD) in the first scenario. The retrieval breaks down into the following steps:

```
<1>[(L-RETRIEVE *DATA) FROM: <- DB-DEPOSIT]
<2>[(L-RETRIEVE *KEY-DATA-SECT*1) FROM: <- (*DATA <1>)]
<3>[(L-RETRIEVE *DATA-SECT*2) FROM: <- (*KEY-DATA-SECT*1 <2>)]
<4>[(L-RETRIEVE *KEY-DATA-SECT*2) FROM: <- (*DATA-SECT*2 <3>)]
```

The retrieval of the DATA can not fail this time because of the initialization, but <2> or <4> can fail. If either fails, the KEY-DATA-SECT*2 must be created. If KEY-DATA-SECT*1 was not found (<2> fails), both the DATA-SECT*2 and the KEY-DATA-SECT*1 must also be created. Without the IF-THENs and IF-FAILs that are known to be TRUE or FALSE and the retrievals that were not done because they had already been done, the resulting CODE-METHODs have the following connections (in simplified notation with the ASK results as the property names):

```
<1>(LISP-VALUE DB-DEPOSIT)
<2>(LISP ASSOC ACCOUNT <1>)
<3>IF-FAIL <2> TEST: (LISP NULL <2>)
  FAIL:      ;nothing just continue at <6>
  ELSE: <4>(LISP CDR <2>)
        <5>(LISP ASSOC ETIME <4>)
<6>IF-FAIL <2> OR <4> TEST: (LISP OR (LISP NULL <2>)
                                (LISP NULL <4>))
  FAIL: <7>(LISP CONS ETIME AMOUNT-M)
        <8>IF-FAIL <2> TEST: (LISP NULL <2>)
          FAIL: <9>(LISP LIST <7>)
                <10>(LISP CONS ACCOUNT <9>)
                <11>(LISP CONS <10> <1>)
                <12>(ASSIGN DB-DEPOSIT <11>)
          ELSE: <13>(LISP CONS <7> <4>)
                <14>(LISP RPLACD <2> <13>)
  ELSE: <15>(LISP RPLACD <5> AMOUNT-M)
```

The coding routine will take this and turn it into part of the code for (CORRECT A-DEPOSIT*1). Variables will be introduced as appropriate. Predicates will be simplified for the Lisp CONDs. And, the ASSIGN will be turned into the appropriate variable assignment.

The resulting programs for this scenario are similar to the programs of the first scenario with a few exceptions. There is an initialization program to set the two free variables for the data bases to NIL. The retrieval from the data bases is accomplished by evaluating the free variables, instead of the use of GET. The resulting programs for initialization and correcting the A-DEPOSITs are as follows:

```
(DEFUN INITIALIZE ()
  (SETQ DB-DEPOSIT NIL)
  (SETQ DB-WITHDRAWAL NIL))

(DEFUN CORRECT-DEPOSIT ()
  (PROG (ETIM TDAY ACCT AMNT KDS1 DS2 KDS2 NKDS2)
      (PRINT 'PLEASE_TYPE_IN_THE_TIME_OF_OCCURRENCE)
      (SETQ ETIM (READ))
      (COND ((NUMBERP ETIM))
          (T (RETURN 'ERROR_NOT_NUMBER)))
      (SETQ TDAY (TODAY))
      (COND ((AND (> ETIM (DIFFERENCE TDAY 30))
                  (< ETIM TDAY)))
          (T (RETURN 'ERROR_NOT_VALID_TIME)))
      (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
      (SETQ ACCT (READ))
      (COND ((NUMBERP ACCT))
          (T (RETURN 'ERROR_NOT_NUMBER)))
      (PRINT 'PLEASE_TYPE_IN_THE_AMOUNT_OF_MONEY)
      (SETQ AMNT (READ))
      (COND ((NUMBERP AMNT))
          (T (RETURN 'ERROR_NOT_NUMBER)))
      (SETQ KDS1 (ASSOC ACCT DB-DEPOSIT))
      (COND (KDS1 (SETQ DS2 (CDR KDS1))
                  (SETQ KDS2 (ASSOC ETIM DS2))))
      (COND ((AND KDS1 KDS2)(RPLACD KDS2 AMNT))
          (T (SETQ NKDS2 (CONS ETIM AMNT))
             (COND (KDS1 (RPLACD KDS1 (CONS NKDS2 DS2)))
                   (T (SETQ DB-DEPOSIT
                            (CONS (CONS ACCT (LIST NKDS2))
                                  DB-DEPOSIT))))))
      (RETURN 'THANK_YOU)))
```

The combination of the predicates for determining whether the time fits within the required bounds is handled by the failure handler when it discovers that both result in the same failure message. The particular style of variable use happens as a result of the coder wanting to turn anything used more than once into a variable.

Section 4 Efficiency Ideas for the Variation

Again, the resulting programs will work, but they are not very efficient in their use of space, because the data bases just continue to grow. Obviously, the restriction on the correction of deposits and withdrawals was made so that old information could be cleared out to keep the data base small. To take advantage of this possibility, it is necessary to use the IDEAs picked up during analysis.

As in the original specification, the first IDEA to arise is to turn the computation of the BALANCE into a subtotal operation. This time some of the SOURCEs for the (SET (AMOUNT-M A-DEPOSIT#3)) have ETIMEs that are different from the A-DEPOSITs they produce. If an A-DEPOSIT or A-WITHDRAWAL is changed or added to the data base after the time when it was included in the subtotal, the subtotal will have to be adjusted to reflect the new AMOUNT-M. The SCHEMA for the implementation of the subtotal anticipates this possibility with a step predicated on some part of the SOURCE being characterized as CHANGE.

```
[(IF-THEN (EXIST [*SOURCE=(SOURCE *ARG=(ULTIMATE-ARG
                                        (AMOUNT: THE)))#2
                  (> (ETIME ACT::)(ETIME *ARG))])
          [(DO (IF-THEN (< (ETIME (ACT (VALUE *SOURCE)))
                           (ETIME ((AMOUNT: THE) INITIAL)))
                        [(FIXUP ((AMOUNT: THE) INITIAL))
                         FOR::: <- (VALUE *ARG)]))
           LOCATION:: <- (DURING *SOURCE)])]
```

That is, if there is some SOURCE that produces data with an ETIME less than the ETIME of the SOURCE program, incorporate a call in all such SOURCEs that checks to see if the ETIME for the changed value is before the time of the subtotal value (the value used as the initial value for the next subtotal calculation). If so, fix up the subtotal value with the changed or newly added value. The SOURCEs of the BALANCE that cause these data are the programs to CORRECT the A-DEPOSITs and to CORRECT the A-WITHDRAWALs. Thus, during the program to CORRECT the A-DEPOSIT when the new value "becomes" the A-DEPOSIT, the following call is included in the program design tree.

```
(IF-THEN (< (ETIME (A-DEPOSIT*1 (CORRECT ...)))
            (ETIME ((BALANCE ACCOUNT*1) INITIAL)))
         [(FIXUP ((BALANCE ACCOUNT*1) INITIAL))
          FOR: <- (AMOUNT (A-DEPOSIT*1 (CORRECT ...)))])
```

When this call is encountered in analyzing the BECOME of the A-DEPOSIT correction, both the conditional and the call to FIXUP must be analyzed. The analysis of the conditional determines that it is not known to always be either true or false. The (ETIME A-DEPOSIT*1) may be either before or after the time when the last subtotal was done. Thus, the program must be equipped with a test to determine which case holds to branch appropriately. The arguments to the test are the ETIME of the corrected A-DEPOSIT and the ETIME of the last initial BALANCE. The ETIME of the corrected A-DEPOSIT is available in the program, so providing it for the test is no problem. The ETIME of the BALANCE value however is a request for something that has not been needed before. To provide it, the assertion of BALANCE values must be augmented. The characterization of the BALANCE handles these problems by making the ETIME part of the CONTENT of ((BALANCE ACCOUNT) INITIAL). Notice that this is a case where the program version of a datum contains more than the original abstraction. As a result, the assertion of the initial BALANCE in ACCEPT will have to include both parts of the contents. Also, the retrieval in CORRECT can know where the ETIME is located for the

conditional. The INTENT for FIXUP claims it is a SOURCE for the value of the balance, but does not claim it is a SOURCE for the ETIME of the BALANCE, so the CORRECT program does not have to ASSERT an ETIME. When it comes time to select a METHOD for doing the FIXUP, the following METHOD is found:

```
[(METHOD (FIXUP AMOUNT))
 OBJECT: <- AMOUNT:1
 FOR: <- AMOUNT:2
 STEPS: [*RETR=(RETRIEVE AMOUNT:2)
        IF-FAIL:: <- (BECOME RESULT:: <- 0)],
       [(BECOME (AMOUNT:2 FIXUP) <-
                (DIFFERENCE (AMOUNT:2 PROTO)
                            (RESULT *RETR)))],
       [*COMP=(COMPUTE (AMOUNT:1 THE))
        REQUIRE:: <-
         (ULTIMATE-ARG:: <- (AND (AMOUNT:1 THE)
                                 (AMOUNT:2 FIXUP)))],
       [(BECOME AMOUNT:1 <- (RESULT *COMP))]]
```

This METHOD creates a new quantity, the FIXUP, which is the difference between the new value of the item being changed and the old value. If there is no existing old value it is just the new value. This quantity "becomes" a specialization of the item being changed, in this case, the A-DEPOSIT. Thus, it is the (A-DEPOSIT*1 FIXUP) and will qualify as an A-DEPOSIT in calculations. With this quantity the METHOD recomputes the amount, in this case the ((BALANCE ACCOUNT*1) INITIAL), and asserts it. Recomputing the amount is done with the arguments of the computation restricted to be just the FIXUP and the old amount. With that restriction the definition of BALANCE can be reduced to just the addition of the FIXUP to the old BALANCE. The sum of the A-WITHDRAWALs is the sum of an empty set, which is known to be zero. The sum of the A-DEPOSITs is the sum of the set which only includes the FIXUP, and is therefore just the value of the FIXUP. The computation requires the retrieval of the value of the initial BALANCE, which will have to be included in the program.

Again, the IDEA to do the subtotal as an INCREMENTAL-TOTAL is discovered

and found to apply. The implementation of this IDEA is the same as in the earlier scenario except that this time there are SOURCEs that have ETIMEs different from their data. The step to handle this circumstance is as foliows:

```
[(DO [(FIXUP (AMOUNT: THE))
      FOR:: <- (VALUE *ARG)])
 LOCATION: <- (DURING *SOURCE=[(SOURCE *ARG=(ULTIMATE-ARG
                                    (AMOUNT: THE)))#2
                          (> (ETIME ACT:::)(ETIME *ARG))])]
```

This is similar to the corresponding step in the SCHEMA for implementing a subtotal, but it is not conditional on the time of the BALANCE and it fixes up the BALANCE rather than the initial value of the BALANCE. Analyzing the step from the subtotal SCHEMA discovers that the condition will always be true, because CORRECT requires that the ETIME of the datum be less than the current time. The analysis requires another characterization of the BALANCE because of the new information, which discovers that the BALANCE and the initial BALANCE are now the same datum and only one need be represented. That means the other fixup step is removed and the ETIMEs of the BALANCE will no longer be needed.

In the earlier scenario by this time all of the storage requirements except for the BALANCE had gone away. In this case, however, the A-DEPOSITs and A-WITHDRAWALs still have USEs in the CORRECT programs. Because of the restriction on the ETIMEs that can be corrected, there is a restriction on the CHANGE characterization of the A-DEPOSITs and A-WITHDRAWALs. The characterization notices this situation and sets up a goal to implement a PRUNE call to clear out items too old to use. The set of A-DEPOSITs that can be used by any program (the union of the USEs) are those whose ETIME is within the last month. Since the elements are required to have after a non-decreasing time, it is safe to get rid of those that no longer qualify. For this a goal is set up to do the PRUNE to all A-DEPOSITs that are not less than a month old. One

difference between this call and the others is that it does not matter where the call gets done as long as it is done regularly. In the absence of any better ideas the programwriter makes a separate program to be run at regular intervals with the call to PRUNE the SETs. This is mainly a utility function, so it makes sense to do it when it does not add to the user's response time. Just as with initialization, there is a special name for such a program, MAINTENANCE, and it is run at regular intervals.

PRUNEing the A-DEPOSITs (or A-WITHDRAWALs) involves going through the elements of the set testing their ETIMEs against the required time. This is similar to the situation encountered in doing a sum of the elements of a set. The actual work of PRUNEing gets done at the data base level. The METHOD for PRUNEing the set results in the call

```
[(PRUNE (DATA-BASE *SET))
 FOR: <- *SET=(SET [A-DEPOSIT#1 ETIME:: <-
                        (SINCE (DIFFERENCE ETIME: 1-MONTH))])].
```

The METHOD for PRUNEing a DATA-BASE is to create an empty DATA, iterate through the old DATA inserting the appropriate parts into the new DATA, and then assert that the new DATA is the DATA of the DATA-BASE. The BODY of the ITERATE is as follows:

```
[... [(IF-THEN  [(NULL-INTERSECTION (MEMBER DATA:1) SET:1)
                 ERR::: <- FALSE])
      ELSE:: [(IF-THEN [(SUBSET (MEMBER DATA:1) SET:1)
                        ERR:::: <- FALSE]
                       [(L-INSERT (SUBPART DATA:1))
                        FOR:::: <- (MEMBER DATA:1)
                        IN:::: <- DATA:2])
              ELSE::: [*PR=(PRUNE (MEMBER DATA:1))
                       FOR:::: <- SET:1],
                      [(L-INSERT (SUBPART DATA:1))
                       FOR:::: <- ((SUBPART DATA:1)(RESULT *PR))
                       IN:::: <- DATA:2]]] ...]
```

If none of the items in the KEY-DATA-SECT (i.e., (MEMBER DATA:1)) are in the desired set, the KEY-DATA-SECT is skipped. If all of the items in the KEY-DATA-SECT are in

the desired set, the KEY-DATA-SECT is inserted into the new DATA:2. Otherwise the KEY-DATA-SECT is pruned, and the result inserted into the new DATA:2. To expand this BODY, it is necessary to find the values of the predicates. Since the KEYs of the KEY-DATA-SECTs are ACCOUNTs, they do not tell very much about the intersection of the desired set and the elements in the KEY-DATA-SECT. WHETHER simply does not know about the two predicates and waiting will not help. Under ordinary circumstances WHETHER would return EITHER in such a case, but the predicates have an ERR property saying if the answer is not known return FALSE. Thus, the BODY turns into the sequence to prune the KEY-DATA-SECT, and then insert it.

To PRUNE a KEY-DATA-SECT is to PRUNE its DATA-SECT. To PRUNE a DATA-SECT involves an iteration analogous to the iteration above. This time the KEYs of the KEY-DATA-SECTs are ETIMEs and WHETHER is able to tell a great deal more about the predicates. Examining NULL-INTERSECTION predicate, the KEY provides the ETIME, so it is either less than a month old or it is not. Therefore, WHETHER answers EITHER. Examining the SUBSET predicate, there is more information. The ETIME is not more than a month old, because this is the ELSE branch of the NULL-INTERSECTION predicate. Therefore the answer is TRUE.

Given the implementation of the A-DEPOSIT and A-WITHDRAWAL data bases as free variables whose values are ASSOCIATION-LISTs of the ETIME and AMOUNT-Ms as before, the program to PRUNE out the data bases comes out as follows:

```
(DEFUN MAINTENANCE ()
 (PROG (DATE)
       (SETQ DATE (DIFFERENCE (TODAY) 30))
       (SETQ DB-DEPOSIT
        (DO ((LST DB-DEPOSIT (CDR LST))
             (NDATD NIL))
            ((NULL LST) NDATD)
            (SETQ NDATD
             (CONS
              (CONS (CAR (CAR LST))
```

```
            (DO ((LST2 (CDR (CAR LST))(CDR LST2))
                 (NDS2D NIL))
                ((NULL LST2) NDS2D)
                (COND ((< (CAR (CAR LST2)) DATE))
                      (T (SETQ NDS2D (CONS (CAR LST2)
                                           NDS2D))))))
          NDATD))))
      (SETQ DB-WITHDRAWAL
       (DO ((LST DB-WITHDRAWAL (CDR LST))
            (NDATW NIL))
           ((NULL LST) NDATW)
           (SETQ NDATW
            (CONS
             (CONS (CAR (CAR LST))
              (DO ((LST2 (CDR (CAR LST))(CDR LST2))
                   (NDS2W NIL))
                  ((NULL LST2) NDS2W)
                  (COND ((< (CAR (CAR LST2)) DATE))
                        (T (SETQ NDS2W (CONS (CAR LST2)
                                             NDS2W))))))
             NDATW))))))
```

It may be noticed that the difference calculation is outside the two sets of DO loops. This is handled by the coder when it processes the constraints on the code and discovers that that piece of code has no need for the element of the list, computes a value, and is needed by both sets of DOs.

The program to CORRECT an A-DEPOSIT now includes the computation to CORRECT the BALANCE utilizing the old A-DEPOSIT if it exists or zero if it does not. It comes out as follows:

```
(DEFUN CORRECT-DEPOSIT ()
  (PROG (ETIM TDAY ACCT AMNT KDS1 DS2
         KDS2 NKDS2 ITEM KDSB NBAL)
        (PRINT 'PLEASE_TYPE_IN_THE_TIME_OF_OCCURRENCE)
        (SETQ ETIM (READ))
        (COND ((NUMBERP ETIM))
              (T (RETURN 'ERROR_NOT_NUMBER)))
        (SETQ TDAY (TODAY))
        (COND ((AND (> ETIM (DIFFERENCE TDAY 30))
                    (< ETIM TDAY)))
              (T (RETURN 'ERROR_NOT_VALID_TIME)))
        (PRINT 'PLEASE_TYPE_IN_THE_ACCOUNT)
        (SETQ ACCT (READ))
        (COND ((NUMBERP ACCT))
              (T (RETURN 'ERROR_NOT_NUMBER)))
```

```
(PRINT 'PLEASE_TYPE_IN_THE_AMOUNT_OF_MONEY)
(SETQ AMNT (READ))
(COND ((NUMBERP AMNT))
      (T (RETURN 'ERROR_NOT_NUMBER)))
(SETQ KDS1 (ASSOC ACCT DB-DEPOSIT))
(COND (KDS1 (SETQ DS2 (CDR KDS1))
            (SETQ KDS2 (ASSOC ETIM DS2))))
(COND ((AND KDS1 KDS2)
       (SETQ ITEM (CDR KDS2))
       (RPLACD KDS2 AMNT))
      (T (SETQ ITEM 0)
         (SETQ NKDS2 (CONS ETIM AMNT))
         (COND (KDS1 (RPLACD KDS1 (CONS NKDS2 DS2)))
               (T (SETQ DB-DEPOSIT
                        (CONS (CONS ACCT (LIST NKDS2))
                              DB-DEPOSIT))))))
(SETQ KDSB (ASSOC ACCT DB-BALANCE))
(SETQ NBAL (PLUS (COND (KDSB (CDR KDSB))(T 0))
                 (DIFFERENCE AMNT ITEM)))
(COND (KDSB (RPLACD KDSB NBAL))
      (T (SETQ DB-BALANCE (CONS (CONS ACCT NBAL)
                                DB-BALANCE))))
(RETURN 'THANK_YOU)))
```

The retrieval of the old ITEM to compute the FIXUP has to wait until the KEY-DATA-SECT*2 has been retrieved, but it also must take place before the KEY-DATA-SECT*2 has been modified. The rest of the additional code is after the old code because that is the first place the ITEM variable is completely determined. Actually, the assignment of KDSB could have come anywhere before the use of it.

This completes the savings account scenarios.

Chapter V
Selected Topics

Now that the scenario has been presented, the reader has seen how all of parts of the programwriter work together to design programs. In the process there was some light shed on a number of topics related to the design of programs and the range of knowledge needed to make the programming decisions. This chapter will explore some of these topics. (A note to the casual reader: This chapter is not important to the understanding of the basic thesis, and therefore can be skipped.) There are five areas that discussed. The first is the nature of the questions EVALUATE and WHETHER have to answer during the designing of a program. The second is a discussion on the recovery from failure, what the options are and how they effect the resulting program. The third is another look at the way IDEAs work. The forth is a more detailed look at how the data base refinements determine program structure. And, the fifth discusses elements of the philosophy of programming embodied in the programwriter.

Section 1 Answering Questions

The two question answerers in the programwriter, EVALUATE for questions of value and WHETHER for predicates, have general mechanisms for handling most of the questions handled during the scenario, but for questions not answerable by those mechanisms there are special routines. These are called through the dispatch mechanism using the pattern of the question. The nature of the program design problem lends itself to this approach for several reasons. For particular kinds of questions there is special knowledge that can be used to aid in answering the question. If the system relied

entirely on a general deduction mechanism, this knowledge would have to be represented in a uniform way along with the rest of the knowledge. This way it is represented procedurally where it is needed. Secondly, there usually is a safe answer to return if a correct answer can not be provided. Thus, the ability to answer all conceivable questions is not critical to producing a program. The result of not answering a question may be the inclusion of code that may not be used or not taking advantage of possible simplifications. One produces more code than necessary. The other produces less efficient code than is possible. Thirdly, even partial answers may be sufficient, if not for the whole question, at least for making some determinations. Finally, some of the questions are answerable at a later time. Questions about the characteristics of the code or ultimate implementation of structures must be delayed until the code or structures have been chosen. It takes specialized knowledge to distinguish between questions that are unanswerable and those that may be answerable later. All of these properties lend support to the use of specialized techniques for answering the *difficult questions*.

Most of the questions that were answered in the scenarios were simple to handle. The questions for WHETHER involved checking for characteristics that were either explicitly TRUE or explicitly FALSE. The questions for EVALUATE involved filling in the variables. However, it is appropriate to take a look at the kinds of situations where special routines were used to handle the questions.

The simplest "special" situation is a characteristic that will be answerable later. Determining whether a result will be NEW, what the initial value of a structure will be, whether an operation will produce an error versus return a result indicating failure, or whether something is a legal value of a construct can all be handled once the Lisp operations and constructs have been determined. It is a service of the target

language model that that information is available on its information structures. The routines answering these questions know that and suspend the questions until the needed constructs are available.

1.1 Failure to Retrieve

One of the more complex problems WHETHER has to handle is determining whether a retrieve will succeed or not. As we saw in the scenario, the proper answering of this question can eliminate many useless branches from a program. Since retrieval can be a multi-stage operation and different retrievals involving some of the same stages may occur in the same program, the proper answer can depend on what else has happened in the program. Also, while the ultimate answers depend on the Lisp implementations, there are many cases where sufficient answers can be produced beforehand.

First, notice that any information that WHETHER can provide about the possibility of failure is useful. TRUE, FALSE, and EITHER will completely determine what branches need to be expanded. However, eliminating possibilities will make it possible to proceed with some part of the refinement. If TRUE can be eliminated (always fail), the branch for handling success can be refined. If FALSE can be eliminated, the failure branch can be refined. If EITHER is eliminated, there will be no need for a test.

To eliminate the possibility of always failing, a check is made to see if there exists a source for the datum. If a source exists, it must be possible for the retrieval to be executed after at least one of the sources has been executed. For example, if there is a conditional on the program that contains the retrieval (as a REQUIRE), it can not be

made FALSE by all of the sources. This determination is made in general by trying each source for the pattern to be retrieved, hypothesizing that it has taken place, checking the requirements on the program doing the retrieving for any predicates that would prevent it from happening, and checking the additional facts at the branch points preceding the retrieval node in the program. Pragmatically, if there is a source for the datum no restrictions on the program and no branches before the retrieval (the normal situation), then the retrieval can succeed (failure is not TRUE).

Part of the ability to produce the rest of the answer depends on the relationships between the three levels of data refinements: the set level operations, the data base level operations, and the Lisp level operations. The set level representations determine what can not be retrieved. If something is not defined at the set level, then until a source for it has been executed (an assertion) there is no way to retrieve it. If a use (a retrieval) can happen before a source has been executed, the use had better be prepared to indicate failure. Data base representations on the other hand are conservative. The Lisp structure used as the implementation of a data base structure must be defined at any time that the data base structure would be defined. If the data base representation claims the data exists, then whatever Lisp representation is used to implement it will be able to return the value. These facts about the representations provide some bounds for what may happen with retrieval.

Deciding between success all of the time and some failure requires the use of the DEFINED and KNOWN characteristics provided by the characterization of the data. For success to be guaranteed, the data must be KNOWN and DEFINED at the final level of refinement. Whether or not it is KNOWN at the data base level, the same will hold at the final level. However, if the data is DEFINED at the set level and (DEFINED NOT) at

the data base level, either could hold at the Lisp level. If the data is DEFINED at the data base level, it must be DEFINED at the Lisp level. If it is (DEFINED NOT) at the set level, it must be (DEFINED NOT) at all levels.

The situation becomes a little more complicated when a data base level retrieval requiring multiple Lisp level retrievals is encountered. What will happen depends on what has happened previously in the program. Given a retrieval in a program occuring after another retrieval, it is known whether or not the first retrieval succeeded. That information can be used in determining what will happen with the second retrieval if they share common refinement steps. A multiple retrieval (one requiring more than one step) will fail if any of its steps fail. Therefore, the information at a second retrieval may be that one of several retrieval steps failed. More precisely, it is known that one of the retrieval steps that can fail has failed. The information about which retrievals can fail is recorded on the node by WHETHER, making it possible to go back and find that information. Therefore, if the second retrieval includes all of the steps that can fail, it too must fail. If it includes some of the steps that can fail, it might fail, but must be tested.

1.2 Determining Change

Another problem faced by WHETHER is determining whether a datum can change. (This occurs inside the CHARACTERIZE module, after which it can be found as a characteristic on the datum.) As mentioned in section III.5, there is a general algorithm to determine the answer. The algorithm is to hypothesize the existence of the datum constrained as indicated by the requirements of the source, and see whether at a later time the datum could be accepted by the source. This algorithm is similar in nature to

the general algorithm mentioned above for determining whether a retrieval can succeed. Consider an example where the restrictions on the programs must be used to determine the answer. Assume that the program to correct deposits was really just a program to accept them late and did not allow corrections for ones that already existed. That is, it has the following restriction:

```
[(REQUIRE ...) <- (NOT (EXIST [A-DEPOSIT#1 ACCOUNT:: <- ACCOUNT*1
                                ETIME:: <- (ETIME ...)]))]
```

It would seem obvious from this restriction that an A-DEPOSIT can not change once it exists because the ACCOUNT and ETIME are the identifiers, but proving it is a little complicated. First, the existence of some A-DEPOSIT that was accepted is proposed:

```
[A-DEPOSIT*1 ACCOUNT: <- ACCOUNT*1
             ETIME: <- ETIME*1
             (NOT (EXIST [A-DEPOSIT#2 ACCOUNT:: <- ACCOUNT*1
                                      ETIME:: <- ETIME*1]))]
```

(The requirement does not further constrain the values that ACCOUNT and ETIME can have, as might happen in other cases.) It is given as the result of a hypothetical event, which will be invalidated when the WHETHER is completed. Then WHETHER is asked about the requirement again. EXIST will answer TRUE to (EXIST A-DEPOSIT*1), because A-DEPOSIT*1 matches and exists by hypothesis. The NOT turns the TRUE into a FALSE, rejecting the A-DEPOSIT*1 and establishing that A-DEPOSITs do not change.

1.3 The Size of a Set

Another question is to ask the size of a set. This happens in the CHARACTERIZE module. The most important feature of a set is its size. Not only is it important to know how a set grows to decide between alternative designs, but it is also important to recognize sets with a fixed number of elements, particularly zero or one, to

simplify the code. The programwriter knows about two kinds of circumstances when the set will be of fixed size.

If the set is restricted to data occurring after some event or set of events, it might have a fixed number of elements. An example is the following set encountered as an argument to (SUBTOTAL ...) in (ACCEPT A-DEPOSIT*1) in the second scenario when the incremental total was being tried:

(SET (AMOUNT-M [A-DEPOSIT*7 [ETIME:: (SINCE (SUBTOTAL ...))]]))

The (SUBTOTAL ...) nodes are in both the (ACCEPT A-DEPOSIT*1) program and the (ACCEPT A-WITHDRAWAL*1) program since the SCHEMA for incremental totals moved the computation to the source. The first check is to see if the set of events includes an event that is a part of the source for the data. In the example case, the A-DEPOSITs and A-WITHDRAWALs are ULTIMATE-ARGs to the computation. (ACCEPT A-DEPOSIT*1) and (ACCEPT A-WITHDRAWAL*1) are the sources for the A-DEPOSITs and A-WITHDRAWALs. Therefore, the events indicated by (SUBTOTAL ...) include the sources for the set. If that criteria is met, there are three questions concerning what happens between the beginning of the limit period (call it the bound) and the point of use of the set of data. That is, between the previous (SUBTOTAL ...)s and the use of the set of AMOUNT-Ms to do a (SUBTOTAL ...).

1) Can there be a source between the program that is the bound and the program in which the set is used? This is a question the data model must answer.

In this case, the answer is no because all of the sources are covered.

2) Is there a BECOME of the source in the bound program after the bound point? This is answered by the argument model.

That is, is the (BECOME A-DEPOSIT*1 ...) after (SUBTOTAL ...) in (ACCEPT A-DEPOSIT*1)? In this case it is not.

3) Is there a BECOME of the use program before the use point? This is also answered by the argument model.

That is, is the (BECOME A-DEPOSIT*1 ...) before (SUBTOTAL ...) in (ACCEPT A-DEPOSIT*1)? It is.

If the first question is true, then the size is not fixed. If the second question is true, then the set is fixed if all of the bound programs are sources. If the second question is false, the set is fixed. If the third question is true, then there is an element in the set in addition to any contributed by question two. (Besides, the additional element already exists and can be used as an argument.) In the example case, the only element is the one asserted just before the (SUBTOTAL ...).

One note about this determination that the set has one element. The intention of the SCHEMA for the incremental total is that the only new element to be included in the computation should be the one in the program. However, it still necessary to establish that that is what happened as a result of the transformation.

Second, if the source of elements in the set has a condition for accepting a new element that an element of the set not exist already, the set can only have one element. Of course if there is no source for elements of the set, the set will remain empty. There are many other kinds of situations where the set could be of fixed size, but they involve less realistic kinds of restrictions on the execution of the programs such as requiring a fixed number of instances of one program between executions of another.

This should give an indication of the level of difficuitly involved in the question answering. Because of the dispatching to special purpose routines, the algorithms for answering these different questions can be used without a general deduction mechanism having to rediscover the appropriate approach each time. Also, the simple context mechanism provided by HYPOTHETICAL events is important in carrying out some of these algorithms.

1.4 Implications of an Answer

In the expansion of an IF-THEN in a METHOD the programwriter uses the answer from a WHETHER to determine what branching is necessary. Sometimes however, the answer really gives the programwriter a choice. A simple example is the whether the result of DELETE will be NEW. DELETE is a Lisp subroutine which removes an element from a list. If the element it removes is the first element, the result is NEW, otherwise it is not. If asked whether the result is NEW, WHETHER should return EITHER. That would mean testing the result to see if it is NEW and either reasserting it or not. If the result is not NEW, reasserting it would not hirt anything. That means the programwriter has the choice of either treating the answer as EITHER or TRUE. If the answer is treated as TRUE, the result may be reasserted when it does not have to be, but the test never has to be done. Situations where an answer other than EITHER is satisfactory in indefinite cases is indicated by an ERR property. These are also the situations when there may be a choice in what to expand.

Another example, is the predicate in the PRUNE METHOD. It does not prune a subsection of the data base if it is known whether all of the elements will be required in the result. There are situations where it is easy to tell that they will be -- the identifiers have passed all of the tests by getting to that subsection. On the other hand when the answer can not be determined from the identifiers, it still may be that because of the data that has come before, they will all be required. Thus, the correct answer is EITHER. However, detecting whether the all of the elements in the subsection will be needed is as hard as doing the pruning. Besides, the pruning will not hurt. Thus, the programwriter should choose the TRUE branch even though the answer is EITHER. In general it is necessary to compare the alternatives to see which is best.

Section 2 Failure Revisited

Once it has been decided that a node can fail to provide the desired result, something must be done to handle the case when failure occurs. Going up the program design tree from such a point, there is usually an IF-FAIL call or a default value that could be used on one of the higher nodes. The current algorithm of the programwriter is to look for the first recommendation for how to proceed. This works but is not the only solution. Any of the suggestions on nodes higher up may provide legitimate alternative solutions. Consider the node structure above the node to retrieve the RECORD full of A-DEPOSITs of the desired ACCOUNT for summing in (STATE (BALANCE ACCOUNT*1)) in the first scenario:

```
<1>(COMPUTE (DIFFERENCE (PLUS ...)(SUM ...)))
<2>(COMPUTE (PLUS ((BALANCE ACCOUNT*1) INITIAL)(SUM ...)))
<3>(COMPUTE (SUM (SET (AMOUNT-M A-DEPOSIT*2))))
<4>(PRODUCE (SET (AMOUNT-M A-DEPOSIT*2)))
<5>(PRODUCE (SET (AMOUNT-M A-DEPOSIT*3)))
<6>(RETRIEVE (SET (AMOUNT-M A-DEPOSIT*3)))
<7>(RETRIEVE (RECORD DB-DEPOSIT)*1)
<8>[(L-RETRIEVE KEY-DATA-SECT*1) FROM: <- DATA*1]
```

Retrieving the RECORD implies that the KEY-DATA-SECT*1 in which it occupies the SECOND-PART must be retrieved. Since the STATE program could be called before there have been any deposits to the account and the set of possible ac[illegible] is not known in advance, the KEY-DATA-SECT*1 may not exist. The retrieval of the KEY-DATA-SECT*1 will fail if it is not there. Since failure means that the RECORD does not exist, something must be done to handle the situation. The RECORD is DEFINED, having a default value of EMPTY, so the obvious solution is to return the representation for empty. Once the RECORD is implemented as a LIST, the empty value becomes NIL. The solution in the scenario is to return NIL. Thus, the code to (RETRIEVE (RECORD ...)) in the scenario is:

```
(COND (KDSD (CDR KDSD))
      (T NIL))
```

KDSD is the KEY-DATA-SECT*1 returned by <8>. If KDSD is NIL, that is not a legitimate KEY-DATA-SECT, but rather an indication that the retrieval failed. Thus, the code says if the KEY-DATA-SECT is found, retrieve the RECORD from it, otherwise use NIL as the RECORD. If the failure handler had checked up the tree a little further, the code would have been different. An empty RECORD means that the SET is empty, but there is a value for the SUM of an empty set if <3> is examined. It is zero. If this fact had been used the code for the SUM (for the parts of <3> following the retrieval of the KEY-DATA-SECT) would be:

```
(COND (KDSD (DO ((LST (CDR KDSD)(CDR LST)) ...)...))
      (T 0))
```

That is, if the KEY-DATA-SECT is found, use the RECORD in the computing the SUM, otherwise just return zero as the value of the SUM. If the failure handler had checked even higher to node <1>, the DIFFERENCE computation would change. (Notice that the PLUS computation would not change. The other argument is already known to be zero, so the METHOD just returns the other argument.) In the case of failure, the SUM is zero and therefore a simpler DIFFERENCE METHOD that returns the negative of the second argument is appropriate. Thus, the code for the DIFFERENCE becomes:

```
(COND (KDSD (DIFFERENCE ...))
      (T (MINUS (DO ...))))
```

At this point there is something else to consider. Below this same DIFFERENCE node there is also a computation of the SUM of the A-WITHDRAWALs, which has the same possibility of failure. If for both the retrieval of the A-DEPOSITs and A-WITHDRAWALs the failure is propagated all of the way to the DIFFERENCE computation, there are four cases for the code to handle instead of two. The special case DIFFERENCE METHODs can be used to advantage, but that is a lot of cases.

In general the propagation of failure results in choices. The difference between propagating the failure to a low level and a high level can be characterized as the difference between a "patch and go" style of programming and a "special case" style. That is, as failure is propagated higher, the failure handling situations from different terminal nodes multiply the number of cases. The differences really are rather minor in the kinds of programs used as examples in the thesis, but in larger cases they could become significant. The differences in the efficiency of the resulting code can also be characterized. The "patch and go" style generally results in less code, because it emphasizes the general path of the program, returning errant situations to that main line as soon as possible. On the other hand, it may result in a longer average execution time, because every case takes the general path. The "special case" style generally results in less execution time on the average, because it separates the program into cases for which simplifications may be applicable. However, there may be more code because each case must be separately handled.

In a large program or a program with a critical path for which the execution time is very important, decisions must be made among the various alternatives. The criteria for decisions is the tradeoff between program space and average time. For example, if a special case amenable to some simplification is actually the situation occurring the majority of the time, it should probably receive special case treatment. On the other hand, if the special cases rarely occur and their path through the main path of the program is not significantly worse than it would be if they were handled as special cases, then they should be returned to the main path. Making such a decision is not difficult, but the criteria is foreign to the standard concept of a space-time tradeoff. Usually such decisions are based entirely on the storage space for data and the execution time, considering the storage required to hold the program itself as free. The cost of program space may be small, but it does need to be considered.

Section 3 Ideas Revisited

The process for improving the basic program has three parts. The first is the search for IDEAs, taking place in the analysis routine. This starts with the each node and the associated data and searches up the genus path for IDEAs. The second part is trying out the ideas. This is done by CHECK-IDEAS by using the context mechanism to isolate the modifications and SCHEMAs to carry out the modifications. The third part is comparing the results of the IDEAs with the basic program and the other IDEA results. There are notes to be made about each of these parts.

3.1 Finding IDEAs

In the scenarios two IDEAs were used, one for converting to a subtotal and one for converting to an incremental total. There are a number of other ways in which IDEAs could have been used to change the basic program. First, consider what might have happened if the SUMs were characterized as TOTALs, as they should have been. Then (SUM (SET (AMOUNT-M ...))) would have caused the following IDEA to be picked up:

```
[(IMPLEMENT (COMPUTE (SUM (SET (AMOUNT-M A-DEPOSIT*2)))))
 AS: <- (SUBTOTAL (SUM (SET (AMOUNT-M A-DEPOSIT*2))))]
```

The analogous IDEA for the SUM of the A-WITHDRAWALs would also be found. The implementation of these IDEAs and the incremental total IDEAs that would follow are very similar to the second scenario except that the result would be two totals for each ACCOUNT instead of one. One total would represent the total of the A-DEPOSITs and one would represent the total of the A-WITHDRAWALs. Thus, the STATE program would have to retrieve both and subtract them. This is not as efficient a solution as the one in the scenario, but if both of those sums were needed separately this would be the best.

Usually, when I think of subtotals or incremental totals, I think of this situation of handling sums. There is really a range of generality for IDEAs. At the most specific end is an IDEA about the count of a set. Rather than counting the elements of a set, the count can be kept as a number, incremented and decremented as elements are added and removed. More general than that is the IDEA to handle all sums incrementally. Still more general is the IDEA for all kinds of TOTALs, found in the scenarios. Beyond this is an IDEA about RECALCULATIONs. A RECALCULATION is a computation, one of whose arguments is the initial value and the rest of whose arguments come into existence since the initial time. In such a case, the computation can be turned into an incremental total. Another generalization but different from the RECALCULATION, is the IDEA that any computation made up exclusively of SUM, PLUS, and DIFFERENCE can be turned into an incremental total. To represent this IDEA would require another term in the argument model to represent all of the operations in a calculation, but with the notion of ULTIMATE-ARG already existing, this is a reasonable addition.

There are even more general ways to think about the problem, although the programwriter does not have the necessary terminology to represent them. The general problem exists in situations where data is generated in programs other than where it is used. The solution is to distribute the computations that will be needed between the source programs and the use programs to minimize the amount of data that has to be transfered between programs. Different kinds of computations can be distributed in different ways. In particular sums and differences can be done incrementally, averages can be done incrementally if two values are saved, maximum and minimum comparisons can be done incrementally, and so forth.

3.2 Implementing an IDEA

The implementing of the IDEAs operates on the program structure in term of the names supplied by the models. Consider the SCHEMA that turns a SUBTOTAL into an INCREMENTAL-TOTAL:

```
[(SCHEMA ((INCREMENTAL-TOTAL AMOUNT)(SUBTOTAL AMOUNT)))
 OBJECT: <- (SUBTOTAL AMOUNT:):
 RESULT: <- (INCREMENTAL-TOTAL AMOUNT:):
 STEPS: (BECOME OBJECT: <- (RETRIEVE (AMOUNT: THE))),
      [(DO *SUBT=(SUBTOTAL AMOUNT:),
          (BECOME (AMOUNT: THE) <- (RESULT *SUBT)))
       LOCATION:: <- (AFTER [(SOURCE *ARG=(ULTIMATE-ARG
                                    (AMOUNT: THE)))#1
                          ((ETIME ACT::) <- (ETIME *ARG))])],
      [(IF-THEN (EXIST [*SOURCE=(SOURCE *ARG=(ULTIMATE-ARG
                                    (AMOUNT: THE)))#2
                     (> (ETIME ACT::)(ETIME *ARG))])
            [(DO [(FIXUP (AMOUNT: THE))
                 FOR:::: <- (VALUE *ARG)])
             LOCATION::: <- (DURING *SOURCE)])],
      (GOAL (ANALYZE (SOURCE (AMOUNT: THE))))]
```

There are four steps this SCHEMA must carry out to turn a SUBTOTAL into an INCREMENTAL-TOTAL. The first is the only one that takes place at the sight passed in as the OBJECT. It is to replace the old SUBTOTAL with a RETRIEVE of the total that will be provided.

The second step is at the other end of the data path for the computation. The location of the SOURCEs of the ULTIMATE-ARGs of the AMOUNT are found with the help of both the data model, finding the right programs and the argument model finding the right places in the programs. The ULTIMATE-ARGs of the computation are supplied by the argument model. The programs that are the SOURCEs of these were recorded by the CHARACTERIZE module, making that information readily available. The ETIME properties of this data is also maintained by the data model -- it was in the

specifications and the INTENTs for the program nodes. Basically, the sources that will match are those done by ACCEPT. Finally, the SOURCE point in the program was found by the argument model when it analyzed the corresponding BECOME. That location (actually set of locations since there may be more than one SOURCE) is given to the interpreter which sets up the appropriate PREMETHOD goals to include the new call. Notice that this is an implicit iteration through all of the matching locations.

The third step takes care of the SOURCEs whose ETIMEs are different from the data they produce. These may change existing data, making it necessary to lookup any previous value and only add in the difference if there is one. The condition on the step checks to see if there are any such SOURCEs. That is a check of the program design tree, rather than something that could turn into a conditional in a program. The actual placing of the code is very similar to the second step.

The final step just assures that the SOURCE points will be analyzed again. That will cause the data to be characterized again, hopefully discovering that it is no longer necessary to store the arguments. Even if this is not the case, it may be possible to prune the data bases as in the final scenario.

The important points to notice about this SCHEMA is how is makes references to places in the program design tree, how it makes additions and changes to the structure, and what kinds of changes it can make. The reference points are supplied given names by the data model, although it takes both the data model and the argument model to track them down. The changes are made relative to the reference points by adding or changing the nodes. It can make any kind of a change for which the models can supply the needed reference points and the conditions can be supplied as predicates. It should also be noted that it would be very hard to produce this kind of program

structure through a standard refinement mechanism. Refinement counts on having a single node at some level which can be refined in the correct way to produce the desired result. In PSI generalizations are made first before the refinement takes place to give maximum opportunity for producing the best refinements. In this case the nodes that need to be changed are not even in the same programs.

3.3 Comparing the Results

Finally, the method for comparing IDEA implementations is based on the class of the storage usage and computation time. I will not argue that the particular choices of classes are the best ones, but this scheme has some advantages over the traditional space-time product. To compute a space-time product it is necessary to have some numbers for the typical rates at which the sets of data will be encountered by the program. It is also necessary to have the complete program structure. Finally, the technique does not differentiate locally undesirable features such as large maximum size, or widely vary times, or slow response time. It is also clear that the human programmer does not bother with calculating space-time products in all but the most critical of cases. The approach taken by the programwriter is to locate the most important space or time features of the programs and compare those. In many cases the most important features in the programs being compared will be the same, so the second or third most important are compared. This way the only thing that has to be known about unrefined parts of the program structure is that it will not make any *important* contributions. This scheme also makes it unnecessary to have specific numerical values for the rates at which information will arrive. As long as the information about the rates is sufficient to pick out the most important features, the information is sufficient.

The three classes used by the programwriter are INCREASING, SAWTOOTH, and CONSTANT. As long as the actual values are comparable, these classes are reasonable. INCREASING is deemed most important because it is unbounded, implying that for later executions of the program the time and space required will become excessive. In fact, if the rate of increase is small, this is not a problem. SAWTOOTH is certainly better than INCREASING, since it is bounded (assuming the slopes are comparable), but it depends on the size whether it is really better than CONSTANT. The determination of an appropriate set of classes depends on the use of the programs and a domain classification of the different significances of rates. For example, in some situations response time is very important, while the time required for a maintenance program is unimportant. In a particular domain certain kinds of data determine the base rate of information flow. Data that occurs more slowly than the base rate could determine a less important class. This kind of information can be used to direct the concern of the programwriter to the critical parts of the program.

Section 4 Data Base Functions

The basic data base functions used by the programwriter are RETRIEVE, ASSERT, INSERT, CREATE, and PRUNE. They interact to maintain the data bases and perform the desired accessing and storing operations. It is useful to take a look at the METHOD for ASSERT to see how this produced the tailored code for the programs. The METHOD for ASSERT used in the scenario is as follows:

```
[(METHOD (ASSERT (PART DATA-BASE)))
 OBJECT: <- *ITEM=((PART DATA-BASE): DATA-BASE:):
 FOR: <- DB-DATUM:
 STEPS: [(IF-THEN (KIND *ITEM DATA)
                  [(L-ASSERT (DATA DATA-BASE:))
                   IN::: <- DATA-BASE:
                   FOR::: <- (DB-DATUM: THE)])
         ELSE:: [*SP=(RETRIEVE (SUPERPART *ITEM))
```

```
IF-FAIL::: *CR=(L-CREATE (SUPERPART *ITEM)),
           [(ASSERT (SUPERPART *ITEM)):1
            FOR:::: <- (RESULT *CR)]],
[*LASS=(L-ASSERT (PART DATA-BASE):)
 IN::: <- (RESULT *SP)
 FOR::: <- (DB-DATUM: THE)],
(IF-THEN
 (CHARACTERIZED (RESULT *LASS) NEW)
 [(ASSERT (SUPERPART *ITEM)):2
  FOR::: <- (RESULT *LASS)])]]
```

This METHOD matches all of the levels of the data base. Thus, its OBJECT is a data base part of a particular data base, e.g., an ITEM of DB-DEPOSIT. The OBJECT specifies what is to be done with the FOR. The FOR is the specific data base datum to be asserted. (DB-DATUM is a substantive characterization for a DATUM or a specific part of a data base.) The algorithm is to do the Lisp level assertion if it is the DATA that is to be asserted. If not, then RETRIEVE the data base part at the next level up in the data base hierarchy. With that, the Lisp level assertion can be done. If the result is NEW, then it will have to be asserted at the next level. If the RETRIEVE fails, the next level of part will have to be created from the data base datum and be asserted.

This METHOD references the appropriate part of the data base structure by using the concept SUPERPART. This is evaluated by a routine supplied by the data model that finds the next higher level in the implemented hierarchy, ignoring EQUIVALENT structures. In that way appropriate operations can be generated for each level of the data base. This METHOD also has calls that are ASSERTs. Thus, it may be used several times for refining in the same program. The ability of the analysis routine to eliminate the unneeded branches and to use the result of retrievals that have already been done are very important in refining using a METHOD such as this. The TRUE clause of the first step (the IF-THEN) is only useful at the top level of the data base, so there is no reason to include it in refinements at any other level. The second branch point is the IF-FAIL.

In the first scenario, this was turned into a conditional in only one of the four uses of the METHOD. The third branch point is like the first -- it is dependent on a condition that is answerable from the implementation and does not result in a test in the program. If the retrievals of the same item were not combined, the DATA in the data base would be retrieved four times in the first scenario.

The possibility of improving the Lisp implementations shows the relation that can exist between the data base level and the Lisp level. One thing that requires additional code for doing an ASSERT is the property NEW on the result of a Lisp level assertion. It means that the resulting structure may no longer be in the containing structure it is supposed to be in. This happens as the result of the Lisp primitive CONS or any other operation that may not return the same structure as it was given. An alternative strategy at the Lisp level is to use HEADED lists in the data base implementation. That is, use a list that has a dummy element at the beginning to provide a permanent handle on the list. Additions are made to the list by replacing the rest of the list by the new set of elements. That way, the result is still in the containing structure. The idea for such an improvement belongs on the pattern for asserting something NEW, which implies some backing up. The scenario is as follows:

The final assertion in the METHOD for ASSERT is conditional on the result being NEW. Therefore, the refinement waits until the data base parts have been implemented before going ahead with the that branch of the METHOD. When the implementation takes place, the RECORD is implemented as a normal LIST because that is the standard way of implementing something with a variable number of elements. As a result the L-ASSERT (actually L-INSERT) that adds the datum to the RECORD is refined as the CODE-METHOD that does CONS. This CODE-METHOD claims the result is NEW.

At this point the conditional in the ASSERT METHOD can be handled. The result is NEW and the ASSERT will have to be done. When the analysis routine adds this new node to the tree, it discovers the IDEA to use a HEADED-LIST for the RECORD instead of a normal LIST. Implementing this IDEA changes the L-INSERT routine to one that does the modification to the HEADED-LIST. The answer to the conditional then changes and the ASSERT no longer has to be done.

All of this effort could be avoided by anticipating the problem in either the ASSERT METHOD or the SCHEMA to implement a RECORD, but it displays an interesting problem of programming. Often, the correct way of doing something does not become apparent until some way is tried. That is, the correct decision at one level depends on the ramifications the various decisions will have at lower levels. The system is deterministic, so it must be possible to produce a predicate that will determine at the higher level which decision to make, but that predicate may be dependent on knowing all of the possible refinements at the lower level. In any realistic program synthesis system it must be possible to learn at a lower level about improvements at a higher level.

Section 5 Philosophy of Programming

The programwriter takes a certain approach to the programming problem. With the scenarios in mind it is useful to examine two features of this approach: the balance between the handling of goals and recognition, and the standardized method approach to problems with many alternatives.

5.1 Goals Versus Recognition

The programwriter operates both by handling goals needed by parts of the design and by recognizing situations that exist. That is, there are elements of both solving problems and forward reasoning. The programwriter during analysis makes the declarations provided by the INTENTs and searches for IDEAs. During characterization it determines a specific set of information about the data. These things are all done with the anticipation that the information will probably be useful, rather than a need to know a specific piece of information. (Characterization is done because information about the data is needed, but the information provided is more than that specifically needed.) On the other hand, such operations as implementations or the making of PREMETHODs happen in response to specific needs. Other systems doing program synthesis have tended toward one extreme or the other. For example, PSI's basic approach is to apply rules, which are in effect antecedent theorems, until the program is complete. This has the effect of producing all possible programs, from which an efficiency module chooses the best. Hacker's basic approach on the other hand was to solve specific problems. It only made changes if something specific, such as a bug or missing code got in the way of a program's ability to operate.

The position taken in this thesis is that there needs to be a balance between these two extremes. There are kinds of information that should be handled when they are noticed, and part of the system needs to be looking out for that kind of information. There are also kinds of information better handled when they are really needed. Information that should be handled when it is noticed has some or all of the following characteristics:

1) There is a high probability that it will be needed during the design process.

2)If it were ignored until it is needed, a search would be required to find it.

3)It would never specifically be required, but might be useful or provide an improvement to the required code.

For instance, the SOURCE for a datum will be required if the datum is ever to be used. It is easy to pick up and declare when it is discovered during analysis, but if the programwriter waited until it was needed all of the nodes in the program design tree would have to be searched. The IDEAs are examples of information that is never actually needed, but can be put to good use if it is available. Information that should wait for specific needs has the opposite characteristics. A specific request will be generated if it is needed, which can be handled in a straight-forward manner. For instance, at some point the programwriter may need to know if a set is a subset of another set. There is no need to generate such information in advance because it is unlikely that any specific piece of such information will ever be needed. Even the PSI system has to have the generation of possible programs controlled by the efficiency module to keep the generation of information from getting out of hand.

One problem in designing a program synthesis system is breaking the problem down into goals specific enough to handle. In a refinement driven system the only specific goal is to match rules to the existing nodes. (Actually, the PSI system also generates goals when a rule will not fit because of a missing refinement, but it could get by without that mechanism.) The model approach of the programwriter is responsible for supplying many of the specific goals that provide the balance between approaches. The checks on consistency with respect to each of the models produce requirements (such as a call to assert a datum or a call to produce a value) that give direction to the refinement.

5.2 Standardized Methods

The programwriter's approach to data bases makes use of a specific form of data bases and particular methods of using those data bases. The rationale for this approach is the reasonable way in which it limits the choices to be made during that part of the designing. The data base structure is flexible and does a reasonable job of fulfilling any requirements the specifications engender. The basic decision made by this data base representation is to turn the identifiers of the datum into levels of the data base. The kinds of representations this eliminates are those that have the elements on a single level. For example, a hash table scheme where all of the identifiers were used simultaneously to produce the hash code would not be considered.

Because of the existence of IDEAs, other kinds of data base schemes are not completely eliminated. They can be used if specific kinds of situations can be identified and recognized where they should be considered. That is, if a simultaneous hash scheme were appropriate in many cases of pattern lookup, the idea to try out the scheme could be put on the generic concept representing pattern lookup. That way, such a scheme is not tried in all cases, slowing down the selection of storage facilities, but it is available for cases where it is likely to be helpful.

This is a technique used by human programmers. A programmer will have a particular methodology with which he feels comfortable, for which he understands the consequences, and for which he can anticipate the pitfalls. He will use this methodology almost to the exclusion of others, increasing his ability to produce code. For situations where he knows that the methodology is not appropriate he will try other less familiar approaches.

The areas where this kind of standard methodology approach is appropriate have some or all of the following characteristics:

1) There are many alternatives, but there is little difference in the efficiency among them.

2) There is one methodology that is clearly better than the others in all but a few recognizable cases.

3) There are pitfalls in the selection process requiring careful consideration of the alternatives.

The first situation exists in most programming languages. There are many available constructs to do the same thing. The human programmer eventually settles on a tool bag of constructs that will handle most situations he normally encounters and ignores the rest of the constructs unless some specific situation arises. Finding a more efficient program may be possible in many situations, but it is not worth the effort. The second situation is true of sorting. Choosing a sorting algorithm is practically an algorithm in itself. There is no reason to spend the time trying various sorting algorithms when with a little checking of the parameters of the problem and following a flowchart such as that in [Martin 1971] will give a method that is hard to beat. The third situation arises in more difficult problems. If the programs being designed involve synchronization, the best approach is to use a known technique for controlling the synchronization, such as semaphores, rather than try to produce a more efficient program by experimentation.

Chapter VI
Conclusions and Recommendations

The primary statement made by this thesis is that programming is organized by integrating the requirements and information gathered from several different views of the program. In the process of exploring the implications of that statement the thesis has become many other things. It is a statement about how a knowledge representation scheme can be used to do programming. It is a statement that a multiple view approach is appropriate for systems to handle complex problems and even for considering the solving of difficult problems. It is a statement about how one part of the programming problem can be done by a computer system. This chapter will take a look at these statements in the perspective of the thesis, with suggestions for extensions and assessments of limitations.

Section 1 Mechanisms

The programwriter uses a knowledge representation scheme based on the Owl-I language, including a hierarchical representation of concepts, a number of different kinds of static information structures, and a dynamic representation of the design effort. These structures have aided the programwriter in a number of ways. It would take another thesis to assess all of the ways that the knowledge representation is used, but the following stand out:

1) The basic form of concepts makes it easy to access the information related to the concept. This is used in many ways. It makes it possible to tie new information to a concept and have it available to any routine that will need that concept.

2) The genus structure makes it possible to find the relationship between related concepts for such purposes as considering all forms of a particular kind of data in a characterization, inheriting properties from more general concepts, and searching for partricular kinds of concepts. Thus, it is possible to use pattern matching to find refinements while keeping the refinements as general as is appropriate.

3) The concepts can be shared by different models. This provides the links necessary for the interfaces through which one model can use the results of another without knowing about the inner workings of the model.

4) The INTENT structures and the METHOD structures provide the information for making assertions and refining in a form that can be easily examined and extended. This way the "rules" for refining are explicit.

5) The program design tree has an open structure which can be examined and changed at any level. This fulfills the needs of the various models for different kinds of facts about the design and maintains the results of all the design work in a form which preserves the purposes.

6) The SCHEMA structures provide the necessary mechanism to make modifications and additions over more than one node in the design tree. This mechanism also benifits from the ability (provided by the models, but resting on the representation) to find locations in the program design tree by their function.

The programwriter provides a measure for the kinds of services a knowledge representation scheme needs to provide to do problems involving design and reasoning. Comparing the Owl-1 representation to other data base systems (such as CONNIVER or a production system), the primary feature distinguishing Owl-1 from the others is the

hierarchical organization of the concepts. That feature was used to limit searches, cut down on redundant information on concepts and provide a representation that keeps track of the relationships between concepts. In future systems the limiting of searches and saving of information space will become increasingly important. Maintaining the relationships between concepts is important for any size system. The alternative is to explicitly (or implicitly with a daemon scheme) discover and record all of the relationships that might be important whenever a new concept is created in the system. Such work is very time consuming and the work involved increases as the size of the data base increases. This problem of the relationship between concepts has been the subject of recent literature under the name "the symbol-mapping problem"[1]

Section 2 Models for Organizing Other Tasks

The organization of a task according to the interactions of several models is applicable to other domains besides programming. As mentioned in the introduction, one obvious place where several models are needed to solve a problem is in diagnosing a problem with a car. You believe there is a problem with the generator. To find plausible next steps in diagnosis, you consider the generator as part of the mechanical system, pointing out the fan belt as something to check, and you consider the generator as part of the electrical system, pointing out the various electrical connections as places to check. The models also assist in suggesting ways of doing the checks. Fan belts need to be under a certain amount of tension, easily tested by flexing the belt. Electrical connections are tested with a meter to measure their conductivity. The models are used as a way to localize the problem to a particular aspect of a subset of the components.

[1] A description of a system to handle this problem is in [Fahlman 1977]. A description of an extension to CONNIVER to answer this need is in [McDermott 1975].

The model then supplies the techniques for operating in the domain of the model. Auto repair manuals are even organized in this manner. They typically have different chapters covering different systems (models) and general diagnostic sections to aid in determining which system might be at fault.

Another place where different models are important is in medical diagnosis and treatment. The human body is made up of many systems, each of which can be viewed separately and can be used to determine plausible places to continue diagnosis or to determine possible effects. The reason that medical students study physiology and anatomy is to develop the models with which to connect the rest of the facts they will need for the other medical skills. More subtle perhaps are some other views a physician must take when treating a patient. The patient is a total functioning human being with needs, abilities, and constraints. Everything the physician does is going to affect that human being in some way. Different measures may be appropriate if the patient is young, old, has a particular handicap, has a hazardous occupation, has responsibilities, has contact with other people, or has a particular psychological makeup. These factors also play a part in diagnosis. If the patient is a worker in a chemical plant, poisoning becomes a likely possibility for many symptoms. Other kinds of views that a physician must consider are in terms of the actions of the agents acting on the patients. If there are drugs being administered, their effects vary over time, over how they are being administered, and over the state of the patient's bodily functions. If a disease is being treated, it has a particular dynamics that must be considered in recognizing its state, prescribing treatment, and analyzing the results of treatment.

Computer systems to aid doctors only take on a small part of the problems of medicine because the field is so large. Even so, it is hard to specify a task that does

not involve more than one view of the patient to a great enough extent that it is difficult to reduce the alternate views to parameters.

Consider another kind of problem. When the coast guard receives word that someone went fishing and has not returned, they have a problem[2] that needs to be considered in terms of a number of models. First, what happened to the fisherman? By asking the people that know him, a reasonable story can be pieced together of what might have happened. He was going after bluefish and had had his motor overhauled last week to be prepared. Already a reasonable scenario can be pieced together. By checking the day's newspapers, it can be determined where bluefish had been seen. The fisherman probably headed for that area, but had some kind of engine trouble, caused by a mistake in overhauling the motor. That would mean that he is now adrift somewhere that the winds and currents could have carried him from his original track. The demands of the problem say that all such scenarios with a reasonable probability of happening be considered.

Now for the other side of the problem. The coast guard has various equipment available for conducting a search. They have aircraft that can cover a large area rapidly. They have helicopters, which are slower but more able to spot a small object. To use this equipment takes time and therefore planning of effective search strategies. The target, a time varying probability distribution, is provided by the model of reasonable scenarios. This must also be supplemented by an assessment of the detection process. If the boat is just adrift, it will probably not be too difficult to spot, unless it is a color that blends in with the water, or there are many other boats in the area of the same description, or the visibility is impaired. Besides planning the search, it

---

[2] I am indebted to Norton Greenfeld for this example. It was to be a project, but lacked funding.

is necessary to be ready to handle the situation once the lost fisherman is found. That may mean having a cutter in the area to tow the boat to shore once it is found, or it may mean having helicopter rescue available to get the fisherman to medical attention as soon as he is found.

Looking at the problem as sketched, there are three main models that determine the action of the coast guard in such a case: the model of probable scenarios, the model of the search process, and the model of the rescue process. The model of the probable scenarios makes use of many smaller units which could be called models of actions: what might have happened after the engine was overhauled, what the winds and currents would do to a drifting motorboat, what "going for bluefish" implies about the intended track, and so forth. These are parts of the model of possible scenarios in the same sense that each of the target language primitives are part of the target language model.

A computer system to plan such a rescue effort needs to be based on these models. The model of the search process needs the appropriate probability distributions from the scenario model. The rescue model needs times from the search model and conditions from the scenario model. The scenario model needs a measure of the difficulties involved from the search model and the rescue model. For example, there is no need in pursuing a scenario with the fisherman falling overboard in the middle of winter in New England because it would not be possible to make a rescue in time. Also, if two scenarios do not differ in their implications for search and rescue, only one need be considered.

The kinds of problems that can benefit from a multi-model approach have the following characteristics:

1)There are several organized views of the component parts of the system around which questions can be answered and direction provided.

2)The problems that need to be solved require information from more than one of the views.

3)The different views break up the problem into managable chunks. That is, what happens inside of one model can be treated separately from the other models at some level.

The obvious candidates are diagnostic problems where there are several "systems" that could be responsible for the troubles, but as we have seen, complex planning problems also have these attributes.

2.1 Models as an Aid in System Design

The use of models can also be an aid in thinking about a problem. Of course taking a particular view of a problem is a common technique to take advantage of more powerful tools in a general domain, but the use of multiple views also offers a way of breaking the problem into parts. If there are multiple views of the problem solving, these can provide alternatives. If there are multiple views of the subjects of the problem solving, as in the case of the programwriter, these offer different perspectives from which information can be gathered to aid in an overall solution. The different views also make it possible to apply the tools available for those views to the appropriate parts.

This use of models also points out potential problems that might occur in a complex problem. Often the same term is used in different models for different concepts. The problem arises in the following way: The concept is the same in the different models for simple cases, but the generalization of the cases in the different

models take different forms. The term can not stand for the two different generalizations in the same problem space. The potential for this problem can be seen in the programwriter. Consider the term *argument*. In the argument and control model the meaning is clear. The arguments to an operation are the items input to that operation, no matter what level of refinement is being discussed. Thus, the arguments to (* A (+ B C)) are A and (+ B C). In the data model the term argument is used with respect to the things that are considered data. In the simple cases, such as (+ B C), the arguments for both models are the same, but for (* A (+ B C)), the data model considers the arguments to be A, B, and C. Knowing that this problem arises because of the different generalizations of a similar concept in two different views, makes the designer aware of the placees where it may happen.

Section 3 Extensions

The general problem of programming includes much more than what the programwriter is able to do. As mentioned in the introduction, it is important to assess how the approach fits into the general problem and the difficulty of extending the approach. The general problem of programming includes such things as problem solving, debugging, modifications of existing programs, acquisition of the specification, learning from a completed effort, and the design of systems. The way these will be discussed is as possible extensions to the program writing system.

One aspect of programming that was not addressed by the programwriter is the problem solving nature of programming. The problems addressed by the programwriter have obvious solutions. That is, there is no particular difficulty in determining how to compute a balance, other than tailoring the computation to the

circumstances. In other programming situations it is necessary to first solve the problem of how the desired effect can be accomplished at all. Handling such a programming problem would fit within the framework provided by the programwriter. For example, a problem solving system such as that of Sacerdoti [3] could be added to the programwriter without too much difficulty. That problem solving system has a very similar refinement and constraint approach. As a problem is expanded, the constraints become apparent. They are then used to determine the restrictions on order for the parts of the program. To the programwriter, this would mean the addition of a model as solving the problem, basically an addition to the argument model.

Some kinds of problem solving involve an overview of what is happening in the program. Often a programmer abandons a particular strategy because the number of cases to be handled separately are exploding. Detecting such a situation would require some kind of overview of the design tree that the programwriter does not have. The combinatorial explosions of time (resulting from lengthy iterations, say) can be detected because the refinement sets up the stategy and the time can be analyzed. However, the explosion of code is a more difficult problem. It is not possible to wait until it is designed before analyzing it, because the design is never finished. A possible strategy is to examine IDEAs when they are discovered for indications of ways to improve the refinement without actually carrying the refinement further. This might avoid some pitfalls, but it does not address the real issue. What is needed is a continual assessment of the amount of work to do and an assessment of how the current strategy is dividing that work. The two gauges on program size are the number of branches at a particular level of the tree and the number of levels. To know when the refinement is getting into trouble it seems to be necessary to have a way of predicting these. In any case, this is a hard problem.

---

[3] As described in [Sacerdoti 1975-1] and [Sacerdoti 1975-2]

The programwriter does not face the issue of debugging except in the sense that the additions made by the models to satisfy their constraints debug the specifications and refinements. Given a program design tree for a program that does not work, there must have been some incorrect information in the specification or a bug in the structures of the programwriter. If the difference between the desired behavior and the actual behavior can be used to discover a specific fault, the whole specification can be run again, but that avoids the issue. How can the program design tree be used to find the source of the fault? Consider a problem: The STATE program reports a negative balance. A debugging system could use the tree structure to determine that the result of the computation was negative because the sum of the withdrawals was larger than the sum of the deposits. The problem could then be traced to the acceptance of withdrawals that will cause the sum of the withdrawals to become too large. This involves a great deal of effort on the part of the debugger, but at least the causal structures are provided by the tree. It is not clear when debugging is the appropriate strategy. In this case the problem would have been solved if the "debugger" had just added the requirement that the balance can not be negative at each of the sources for the balance and called the programwriter again.

The problem of modifying existing programs has some requirements that the programwriter would not be able to satisfy. The first problem is recognition of the purposes of the parts of the program. The programwriter would have a tough time recognizing its own programs because the statements may not even end up in the same order as the terminal nodes of the tree. The refinement structures have some of the basic information for recognition, although it is not organized for recognition. The steps would provide the patterns to be matched against the existing operations, but the matching process would be complicated by several facts. Not all of the steps need to be

represented in the resulting code. If the conditions can be determined in advance, whole branches are not required. These possibilities would have to be considered in the attempt to match. Also, the code represents the evaluated call. Whether a particular call could result in a particular construct depends on what else is in the tree. Recognizing a program also implies a kind of completeness in the constructs that the programwriter does not have. The programwriter gets along fine with a set of constructs it is comfortable with. It does not know about all constructs or even all about the constructs it uses.

One place the programwriter could be extended is in acquiring a specification. A number of other systems have addressed this problem[4], taking a number of different approaches. The programwriter has a particular approach to specifications that has merits. Most systems separate the acquisition of the specification from the rest of the program synthesis. This means that the specification must be complete when it leaves that phase. If the specification is viewed as an abstraction of the program, that abstraction must totally describe the program. This restriction is usually relaxed somewhat by letting the user answer questions later, but this limits how "abstract" the abstraction can be. The approach used by the programwriter is to handle the specification through the domain model. The domain model is ready to supply any details needed during refinement that were left out of the specification. Having the definitions, relationships and default values prevents the user from having to do part of the job of the programwriter in anticipating what will be needed.

The problem of learning from previous efforts is another problem the programwriter is not organized to handle. One kind of learning is the generation of

[4]For example, PSI [Green 1976] and Balzer's project at USC-ISI [Balzer 1975]

subroutines. The efforts of the programwriter are actually in the opposite direction. A subroutine is a generalization of a piece of code. The programwriter seeks to make its code as specific as possible. There is a place in programming for both kinds of effort. Making code specific makes it more efficient. Making code general makes less of it. My feeling is that the code should be made efficient first. Then those sections that can be generalized without adversely affecting the efficiency can be turned into subroutines (assuming there is some reason to). Recognizing two sections of code that could be calls to a subroutine should be no more difficult using the design tree than using a macro expansion language such as that of Hacker ([Sussman 1973]). The nodes have the same kind of information attached to them and can find any similar nodes by searching the genus structure in Owl-1.

Another kind of learning is learning from previous detours. This would make a nice addition to the programwriter, possibly in the form of suggestion lists for choosing implementations, ideas, and refinements. For example, the example of using lists with dummy elements at the beginning to prevent having to reassert the results could be attached as a suggestion from the ASSERT METHOD to the IMPLEMENT goal for the record. It belongs in the ASSERT METHOD because that is the highest place in the design tree that changes as a result of the IDEA implementation.

Finally, the programwriter deals with small problems. It is more important to be able to design systems. That is, sets of programs that use other programs as primitives. A system is designed in pieces. It is split into modules that can be individually designed and the overall structure is designed with the modules as primitives. The relationship between the modules is analogous to the I/O behavior in the programwriter. The programwriter would fit right in designing the modules, but designing

the overall structure is different. Suddenly, the target language can change its primitives to improve the design. At first this sounds good, but it may also mean a great deal of difficulty in answering questions about the primitives. The answers to many questions become conditional on certain features of the primitives which may change. The main problem, as it is with human programmers, is deciding on the modules. To make a reasonable system, the modules need to be at a level where the work effort is fairly divided between modules. What that requires the system to know about the problem, I can only begin to guess.

All of these extensions to the programwriter are to handle more parts of the programming problem. That does not mean I think it does a complete job of handling the part it does. Several extensions were mentioned in the body of the thesis, trying other possibilities in recovering from failure, handling the problems of ordering, trying other possibilities when there is a choice from WHETHER, and so forth. The most obvious direction the programwriter needs to be extended is in the number and variety of IDEAs it has to work with. The next logical step is a recognition hierarchy of IDEAs. That is, an IDEA can be used to check for a characteristic of part of the program. For example: it is made up entirely of additions and subtractions; it adds elements to a list in date order; it is a recalculation; and so forth. These characteristics could then be the basis for finding other IDEAs. This would mean that IDEAs could be found from more global properties of the developing design.

Section 4 A Last Look at Models

The programwriter used five models of the program to direct the refinement process, the domain model, the argument and control model, the data model, the I/O

model, and the target language model. These models provided the needed insights to handle the programming problems in a reasonable way. Are they the right models for some other programming problem? For most kinds of programming, I believe so. They may look different in other kinds of programming, but there must be something to correspond to these in any programming area. The domain model is a statement of how the program relates to the task outside of the computer that it is handling. Each program has a meaning, if only in terms of the abstractions used as primitives of a higher level of a system's design. Whatever the meaning of the program, the concepts that make up that meaning will form some kind of model that can be used to help the program to maintain the meaning. The argument and control model must exist to coordinate any program where order makes a difference or any primitive needs information from another. The data model is important in any program that will have to maintain data between program executions or will have to maintain an internal representation of data received. The I/O model maintains the relationship between the program and the external environment, whether this is a human or another program which must be treated as a black box. The target language model is needed to maintain the relationship between whatever primitives are used for the writing of the system. The may be a normal programming language or some higher level abstraction which will in turn be programmed.

The companion question is whether other models will be needed in other situations. The answer is yes for two reasons. First, there are features of a program, such as its run time behavior in a real time environment, that do not fit in another model. More importantly, it may be appropriate to break some of the models into parts for particular kinds of problems. For designing a system to do medical diagnosis, the domain has several models besides the other models of the programs. Therefore, it might be

appropriate to consider more than one domain model in designing some parts of the system. Often there are several different kinds of I/O going on in a program, I/O with the user, I/O with other programs, and I/O with hardware devices. If these do not interact with one another, it is appropriate to utilize separate models because the details of the interaction constraints are quite different for the different kinds of I/O. Similarly for the other models there are logical breakdowns that can be made in different situations. The criteria for making such a breakdown is that parts can be treated separately and that they behave in a sufficiently different way that they can benefit from a different model.

Bibliography

Balzer, Robert, "Automatic Programming," in *Annual Technical Report, May 1974 - June 1975,* ISI/SR-75-3, USC-Information Sciences Institute, Marina del Rey, California, September 1975.

Balzer, Robert, *Automatic Programming*, RR-73-1, USC-Information Sciences Institute, Marina del Rey, California, September 1972.

Balzer, Robert, "Specification Acquisition from Experts," in *Annual Technical Report, July 1975 - June 1976,* ISI/SR-76-6, USC-Information Sciences Institute, Marina del Rey, California, July 1976.

Barstow, David R., "A Knowledge-based System for Automatic Program Construction," Advance Papers of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August 1977.

Barstow, David R. and Elaine Kant, "Observations on the Interaction Between Coding and Efficiency Knowledge in the Psi Program Synthesis System," Second International Conference on Software Engineering, San Francisco, California, October 1976.

Biermann, A. W. and R. Krishnaswamy, "Constructing Programs from Example Computations," CISRC-TR-74-5, Ohio State University, Columbus, Ohio, August 1974.

Biermann, Alan W., "Approaches to Automatic Programming," in *Advances in Computers*, Vol. 15, ed. Morris Rubinoff and Marshall C. Yovits (New York: Academic Press, 1976), pp. 1-63.

Bosyj, Michael, *A Program for the Design of Procurement Systems*, TR-160, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1976.

Brooks, Ruven, *A Model of Human Cognitive Behavior in Writing Code for Computing Programs*, TR-75-1084, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1975.

Brown, A. L., *Qualitative Knowledge, Causal Reasoning, and the Localization of Failures*, AI-TR-362, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1977.

Brown, Gretchen P., *A Framework for Processing Dialogue*, TR-182, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1977.

Buchanan, J. R., *A Study in Automatic Programming*, AI Memo 245, Artificial Intelligence Laboratory, Stanford University, Stanford, California, May 1974.

Buchanan, J. R., and D. C. Luckham, *On Automating the Construction of Programs*, AI Memo

236, Artificial Intelligence Laboratory, Stanford University, Stanford, California, May 1974.

Darlington, Jared, "Automatic Program Synthesis in Second-Order Logic," Advance Papers of the Third International Joint Conference on Artificial Intelligence, Stanford, California, August 1973.

Davis, Randall, *Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases*, AI Memo 283, Artificial Intelligence Laboratory, Stanford University, Stanford, California, July 1976.

Dijkstra, E. W., "Notes on Structured Programming," in *Structured Programming*, O. J. Dahl, et al., Academic Press, London, 1972.

Fahlman, Scott, "A Hypothesis-Frame System for Recognition Problems," PhD Proposal, Working Paper 57, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, Dec 1973.

Fahlman, Scott, "Thesis Progress Report: A System for Representing and Using Real-World Knowledge," Memo 331, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1975.

Goldberg, P. C., "The Future of Programming for Non-programmers," Report RC 5975, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, May 1976.

Goldstein, Ira P., *Understanding Simple Picture Programs*, AI-TR-294, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1974.

Green, Cordell, "The Design of the PSI Program Synthesis System," Second International Conference on Software Engineering, San Francisco, California, October 1976.

Green, Cordell, and David Barstow, *A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program-Understanding System*, AI-Memo-258, Artificial Intelligence Laboratory, Stanford University, Stanford, California, January 1975.

Green, Cordell, and David Barstow, "Some Rules for the Automatic Synthesis of Programs," Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 1975.

Green, Cordell, et. al., "Progress Report on Program-Understanding Systems," AI-Memo-240, Artificial Intelligence Laboratory (also STAN-CS-74-444, Computer Sciences Department), Stanford University, Stanford, California, August 1974.

Hammer, Michael, *A New Grammatical Transformation into Detrministic Top-Down Form*, TR-119, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1974.

Hardy, Steven, "Synthesis of Lisp Functions from Examples," Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 1975.

Hawkinson, Lowell, "The Representation of Concepts in OWL," Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 1975.

Heidorn, G. E., "Automatic Programming Through Natural Language Dialogue: A Survey," *IBM Journal of Research and Development*, Vol. 20, No. 4 (July 1976), p. 302.

Hewitt, Carl, *Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot*, AI-TR-258, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1972.

Kuipers, Benjamin J., "A Frame for Frames: Representing Knowledge for Recognition," AI Memo 322, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1975.

Long, William, "Question Answering in Owl," Automatic Programming Group Internal Memo, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1975.

Manna, Z., and R. J. Waldinger, "Toward Automatic Program Synthesis," *Communications of the ACM* 14 No. 3 (March 1971), pp. 151-165.

Manna, Z., and R. J. Waldinger, "Knowledge and Reasoning in Program Synthesis," Technical Note 98, Stanford Research Institute, Menlo Park, California, November 1974.

Mark, W. S., *The Reformulation Model of Expertise*, TR-172, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1976.

Martin, W. A., *OWL, A System for Building Expert Problem Solving Systems Involving Verbal Reasoning*, Course Notes for 6.871, unpublished, Massachusetts Institute of Technology, Cambridge, Massachusetts, Fall 1974.

Martin, W. A., "Sorting," *ACM Computing Surveys* 3 No. 4 (December 1971), pp. 147-174.

Martin, W. A., "A Theory of English Grammar," (in preparation), 1977.

McCune, Brian P., "The PSI Program Model Builder: Synthesis of Very High-Level Programs," Proceedings of the ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, Rochester, New York, August 1977.

McDermott, D. V., "Very Large Planner-Type Data Bases," AI Memo 339, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1975.

McDermott, D. V. and G. J. Sussman, "The Conniver Reference Manual," AI Memo 259a, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1974.

Minsky, Marvin, "A Framework for Representing Knowledge," AI Memo 306, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1974.

Moon, David, *MACLISP Reference Manual*, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1974.

Morgenstern, M. L., *Automated Design and Optimization of Management Information System Software*, PhD Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1976.

Naur, Peter, "An Experiment on Program Development," Bit 12, (1972) pp. 347-365.

Rich, Charles and Howard E. Shrobe, *Initial Report on a Lisp Programmer's Apprentice*, AI TR-354, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1976.

Ruth, Gregory R., "Automatic Design of Data Processing Systems," Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, January 1976.

Ruth, Gregory R., "Protosystem I: An Automatic Programming System Prototype," LCS-TM-72, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, July 1976.

Sacerdoti, Earl D., "A Structure for Plans and Behavior," Technical Note 109, Stanford Research Institute, Menlo Park, California, August 1975.

Sacerdoti, Earl D., "The Nonlinear Nature of Plans," Technical Note 101, Stanford Research Institute, Menlo Park, California, January 1975.

Srinivasan, Chitoor, "The Architecture of Coherent Information System: A General Problem Solving System," *IEEE Transactions on Computers* C-25 No. 4 (April 1976), pp. 390-402.

Srinivasan, Chitoor, "Programming Over a Knowledge Base: The Basis for Automatic Programming," SOSAP-TM-4, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, December 1973.

Shaw, David E., William R. Swartout, and C. Cordell Green, "Inferring Lisp Programs from Examples," Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 1975.

Spitzen, Jay M., "Approaches to Automatic Programming," TR 17-74, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1974.

Summers, Phillip D., "A Methodology for LISP Program Construction from Examples," Report RC 5909, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, March 1976.

Summers, Phillip D., "Program Construction from Examples," Report RC 5637, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, September 1975.

Sunguroff, Alex, "Owl Interpreter Reference Manual," Automatic Programming Group Internal Memo, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1976.

Sussman, G. J., *A Computational Model of Skill Acquisition*, AI TR-297, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1973.

Swartout, William R., *A Digitalis Therapy Advisor with Explanations*, TR-176, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1977.

Szolovits, P., L. Hawkinson, and W. A. Martin, "An Overview of OWL, a Language for Knowledge Representation," LCS-TM-86, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1977.

Waldinger, Richard, "Achieving Several Goals Simultaneously," Technical Note 107, Stanford Research Institute, Menlo Park, California, July 1975.

Waldinger, R. J. and K. N. Levitt, "Reasoning About Programs," Technical Note 86, Stanford Research Institute, Menlo Park, California, March 1974.

Wegbreit, Ben, "The Synthesis of Loop Predicates," *Communications of the ACM* Vol. 17 No. 2 (February 1974), pp. 102-112.

Wirth, N., "Program Development by Stepwise Refinement," *Communications of the ACM* Vol. 14 No. 4 (April 1971), pp. 221-227.

Official Distribution List

Defense Documentation Center
Cameron Station
Alexandria, Va 22314 — 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, Va 22217 — 2 copies

Office of Naval Research
Code 102IP
Arlington, Va 22217 — 6 copies

Office of Naval Research
Code 200
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 455
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 458
Arlington, Va 22217 — 1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, Ma 02210 — 1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, Il 60605 — 1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, Ca 91106 — 1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003 — 1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375 — 6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380 — 1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, Ca 92152 — 1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Mathematics Department
Bethesda, Md 20084 — 1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy